

GaPiL

Guida alla Programmazione in Linux

Simone Piccardi

28 settembre 2020

Copyright © 2000-2019 Simone Piccardi. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being “Un preambolo” in “Prefazione”, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Indice

Un preambolo	xiii
Prefazione	xv
I Programmazione di sistema	1
1 L'architettura del sistema	3
1.1 Una panoramica	3
1.1.1 Concetti base	3
1.1.2 Il kernel e il sistema	4
1.1.3 <i>System call</i> e funzioni di libreria	5
1.1.4 Un sistema multiutente	6
1.2 L'architettura di file e directory	7
1.2.1 Una panoramica generale	7
1.2.2 La risoluzione del nome di file e directory	9
1.2.3 I tipi di file	10
1.2.4 Le due interfacce per l'accesso ai file	11
1.3 Gli standard	12
1.3.1 Lo standard ANSI C	13
1.3.2 I tipi di dati primitivi	13
1.3.3 Lo standard System V	14
1.3.4 Lo “ <i>standard</i> ” BSD	15
1.3.5 Gli standard IEEE – POSIX	15
1.3.6 Gli standard X/Open – Opengroup – Unix	17
1.3.7 Il controllo di aderenza agli standard	18
2 L'interfaccia base con i processi	23
2.1 Esecuzione e conclusione di un programma	23
2.1.1 L'avvio e l'esecuzione di un programma	23
2.1.2 Chiamate a funzioni e <i>system call</i>	25
2.1.3 La terminazione di un programma	28
2.1.4 Esecuzione di funzioni preliminari all'uscita	29
2.1.5 Un riepilogo	30
2.2 I processi e l'uso della memoria	31
2.2.1 I concetti generali	31
2.2.2 La struttura della memoria di un processo	32
2.2.3 Allocazione della memoria per i programmi C	34

2.2.4	Il controllo della memoria virtuale	38
2.2.5	Gestione avanzata dell'allocazione della memoria	42
2.3	Argomenti, ambiente ed altre proprietà di un processo	44
2.3.1	Il formato degli argomenti	45
2.3.2	La gestione delle opzioni	45
2.3.3	Le variabili di ambiente	47
2.4	Problematiche di programmazione generica	50
2.4.1	Il passaggio di variabili e valori di ritorno nelle funzioni	51
2.4.2	Il passaggio di un numero variabile di argomenti	52
2.4.3	Il controllo di flusso non locale	54
2.4.4	La <i>endianness</i>	56
3	La gestione dei processi	59
3.1	Le funzioni di base della gestione dei processi	59
3.1.1	L'architettura della gestione dei processi	59
3.1.2	Gli identificatori dei processi	61
3.1.3	La funzione <i>fork</i> e le funzioni di creazione dei processi	62
3.1.4	La conclusione di un processo	69
3.1.5	Le funzioni di attesa e ricezione degli stati di uscita	71
3.1.6	La famiglia delle funzioni <i>exec</i> per l'esecuzione dei programmi	77
3.2	Il controllo di accesso	81
3.2.1	Gli identificatori del controllo di accesso	81
3.2.2	Le funzioni di gestione degli identificatori dei processi	83
3.2.3	Le funzioni per la gestione dei gruppi associati a un processo	88
3.3	La gestione della priorità dei processi	89
3.3.1	I meccanismi di <i>scheduling</i>	89
3.3.2	Il meccanismo di <i>scheduling</i> standard	91
3.3.3	Il meccanismo di <i>scheduling real-time</i>	94
3.3.4	Il controllo dello <i>scheduler</i> per i sistemi multiprocessore	99
3.3.5	Le priorità per le operazioni di I/O	103
3.4	Problematiche di programmazione <i>multitasking</i>	105
3.4.1	Le operazioni atomiche	106
3.4.2	Le <i>race condition</i> ed i <i>deadlock</i>	106
3.4.3	Le funzioni rientranti	107
4	La gestione di file e directory	109
4.1	L'architettura della gestione dei file	109
4.1.1	Il funzionamento del <i>Virtual File System</i> di Linux	109
4.1.2	Il funzionamento di un filesystem Unix	113
4.1.3	Alcuni dettagli sul filesystem <i>ext2</i> e successori	115
4.1.4	La gestione dell'uso dei filesystem	117
4.2	La gestione di file e directory	127
4.2.1	La gestione dei nomi dei file	127
4.2.2	La creazione e la cancellazione delle directory	134
4.2.3	Lettura e scansione delle directory	136
4.2.4	La directory di lavoro	143
4.2.5	La creazione dei <i>file speciali</i>	145
4.2.6	I file temporanei	147
4.3	La manipolazione delle caratteristiche dei file	150

4.3.1	La lettura delle caratteristiche dei file	150
4.3.2	I tipi di file	152
4.3.3	Le dimensioni dei file	154
4.3.4	I tempi dei file	155
4.4	Il controllo di accesso ai file	160
4.4.1	I permessi per l'accesso ai file	160
4.4.2	I bit dei permessi speciali	164
4.4.3	Le funzioni per la gestione dei permessi dei file	165
4.4.4	La gestione della titolarità dei file	169
4.4.5	Un quadro d'insieme sui permessi	170
4.5	Caratteristiche e funzionalità avanzate	170
4.5.1	Gli attributi estesi	171
4.5.2	Le <i>Access Control List</i>	176
4.5.3	La gestione delle quote disco	187
4.5.4	La gestione dei chroot	193
5	La gestione dell'I/O su file	197
5.1	L'interfaccia dei <i>file descriptor</i>	197
5.1.1	I <i>file descriptor</i>	197
5.1.2	Apertura, creazione e chiusura di un file	199
5.1.3	La gestione della posizione nel file	207
5.1.4	Le funzioni per la lettura di un file	209
5.1.5	Le funzioni per la scrittura di un file	211
5.2	Caratteristiche avanzate	212
5.2.1	La gestione dell'accesso concorrente ai files	212
5.2.2	La duplicazione dei file descriptor	214
5.2.3	Le funzioni di sincronizzazione dei dati	216
5.2.4	Le <i>at-functions</i> : <i>openat</i> e le altre	217
5.2.5	Le operazioni di controllo sui file descriptor	231
5.3	L'interfaccia standard ANSI C	240
5.3.1	I <i>file stream</i>	240
5.3.2	Le modalità di bufferizzazione	242
5.3.3	Apertura e chiusura di uno <i>stream</i>	243
5.3.4	Gestione dell'I/O e posizionamento su uno <i>stream</i>	245
5.3.5	Input/output non formattato	248
5.3.6	Input/output formattato	254
5.4	Funzioni avanzate	258
5.4.1	Le funzioni di controllo	258
5.4.2	Il controllo della bufferizzazione	259
5.4.3	Gli <i>stream</i> e i <i>thread</i>	261
6	La gestione del sistema, del tempo e degli errori	265
6.1	La gestione di caratteristiche e parametri del sistema	265
6.1.1	Limiti e caratteristiche del sistema	265
6.1.2	Limiti e caratteristiche dei file	269
6.1.3	I parametri del kernel ed il filesystem <i>/proc</i>	270
6.2	La gestione del sistema	273
6.2.1	La gestione delle informazioni su utenti e gruppi	273
6.2.2	Il registro della <i>contabilità</i> degli utenti	276

6.2.3	La gestione dello spegnimento e del riavvio	279
6.3	Il controllo dell'uso delle risorse	282
6.3.1	L'uso delle risorse	282
6.3.2	Limiti sulle risorse	284
6.3.3	Le informazioni sulle risorse di memoria e processore	288
6.3.4	La <i>contabilità</i> in stile BSD	290
6.4	La gestione dei tempi del sistema	291
6.4.1	La misura del tempo in Unix	291
6.4.2	La gestione del <i>process time</i>	293
6.4.3	Le funzioni per il <i>calendar time</i>	294
6.4.4	La gestione delle date.	298
6.5	La gestione degli errori	302
6.5.1	La variabile <i>errno</i>	302
6.5.2	Le funzioni <i>strerror</i> e <i>perror</i>	303
6.5.3	Alcune estensioni GNU	305
7	I segnali	307
7.1	Introduzione	307
7.1.1	I concetti base	307
7.1.2	Le <i>semantiche</i> del funzionamento dei segnali	308
7.1.3	Tipi di segnali	309
7.1.4	La notifica dei segnali	310
7.2	La classificazione dei segnali	311
7.2.1	I segnali standard	311
7.2.2	I segnali di errore	313
7.2.3	I segnali di terminazione	314
7.2.4	I segnali di allarme	315
7.2.5	I segnali di I/O asincrono	315
7.2.6	I segnali per il controllo di sessione	316
7.2.7	I segnali di operazioni errate	317
7.2.8	Ulteriori segnali	317
7.2.9	Le funzioni <i>strsignal</i> e <i>psignal</i>	318
7.3	La gestione di base dei segnali	319
7.3.1	Il comportamento generale del sistema	319
7.3.2	L'installazione di un gestore	321
7.3.3	Le funzioni per l'invio di segnali	322
7.3.4	Le funzioni di allarme ed i <i>timer</i>	324
7.3.5	Le funzioni di pausa e attesa	327
7.3.6	Un esempio elementare	329
7.4	La gestione avanzata dei segnali	330
7.4.1	Alcune problematiche aperte	331
7.4.2	Gli <i>insiemi di segnali</i> o <i>signal set</i>	333
7.4.3	La funzione <i>sigaction</i>	335
7.4.4	La gestione della <i>maschera dei segnali</i> o <i>signal mask</i>	340
7.4.5	Criteri di programmazione per i gestori dei segnali	343
7.5	Funzionalità avanzate	344
7.5.1	I segnali <i>real-time</i>	344
7.5.2	La gestione avanzata delle temporizzazioni	347
7.5.3	Ulteriori funzioni di gestione	356

7.5.4	I <i>pidfd</i> e l'invio di segnali <i>race-free</i>	358
8	Terminali e sessioni di lavoro	359
8.1	L'interazione con i terminali	359
8.1.1	Il <i>job control</i>	359
8.1.2	I <i>process group</i> e le <i>sessioni</i>	361
8.1.3	Il terminale di controllo e il controllo di sessione	363
8.1.4	Dal login alla shell	366
8.1.5	Interazione senza terminale: i <i>demoni</i> ed il <i>syslog</i>	368
8.2	L'I/O su terminale	375
8.2.1	L'architettura dell'I/O su terminale	375
8.2.2	La gestione delle caratteristiche di un terminale	376
8.2.3	La gestione della disciplina di linea.	387
8.2.4	Operare in <i>modo non canonico</i>	389
8.3	La gestione dei terminali virtuali	390
8.3.1	I terminali virtuali	390
8.3.2	Allocazione dei terminali virtuali	390
9	La gestione avanzata dei processi	391
9.1	La gestione avanzata della sicurezza	391
9.1.1	La gestione delle <i>capabilities</i>	391
9.1.2	La gestione del <i>Secure Computing</i>	407
9.1.3	Altre funzionalità di sicurezza.	407
9.2	Funzioni di gestione e controllo	409
9.2.1	La funzione <i>prctl</i>	409
9.2.2	La funzione <i>ptrace</i>	414
9.2.3	La funzione <i>kcmp</i>	414
9.3	La gestione avanzata della creazione dei processi	414
9.3.1	La <i>system call clone</i>	415
9.3.2	La gestione dei <i>namespace</i>	418
9.4	Funzionalità avanzate e specialistiche	418
9.4.1	La gestione delle operazioni in virgola mobile	418
9.4.2	L'accesso alle porte di I/O	418
10	La gestione avanzata dei file	419
10.1	Il <i>file locking</i>	419
10.1.1	L' <i>advisory locking</i>	419
10.1.2	La funzione <i>flock</i>	420
10.1.3	Il <i>file locking</i> POSIX	422
10.1.4	Gli <i>open file descriptor locks</i>	430
10.1.5	Il <i>mandatory locking</i>	430
10.2	L'I/O <i>multiplexing</i>	431
10.2.1	La problematica dell'I/O <i>multiplexing</i>	432
10.2.2	Le funzioni <i>select</i> e <i>pselect</i>	433
10.2.3	Le funzioni <i>poll</i> e <i>ppoll</i>	437
10.2.4	L'interfaccia di <i>epoll</i>	439
10.2.5	La notifica di eventi tramite file descriptor	446
10.3	L'accesso <i>asincrono</i> ai file	455
10.3.1	Il <i>Signal driven I/O</i>	455

10.3.2	I meccanismi di notifica asincrona.	457
10.3.3	L'interfaccia POSIX per l'I/O asincrono	467
10.4	Altre modalità di I/O avanzato	472
10.4.1	File mappati in memoria	472
10.4.2	I/O vettorizzato: <i>readv</i> e <i>writew</i>	482
10.4.3	L'I/O diretto fra file descriptor: <i>sendfile</i> e <i>splice</i>	484
10.4.4	Gestione avanzata dell'accesso ai dati dei file	491
10.4.5	Altre funzionalità avanzate	495
11	L'intercomunicazione fra processi	497
11.1	L'intercomunicazione fra processi tradizionale	497
11.1.1	Le <i>pipe</i> standard	497
11.1.2	Un esempio dell'uso delle <i>pipe</i>	499
11.1.3	Le funzioni <i>popen</i> e <i>pclose</i>	502
11.1.4	Le <i>pipe</i> con nome, o <i>fifo</i>	504
11.1.5	La funzione <i>socketpair</i>	510
11.2	L'intercomunicazione fra processi di System V	511
11.2.1	Considerazioni generali	511
11.2.2	Il controllo di accesso	513
11.2.3	Gli identificatori ed il loro utilizzo	514
11.2.4	Code di messaggi	516
11.2.5	I semafori	525
11.2.6	Memoria condivisa	536
11.3	Tecniche alternative	548
11.3.1	Alternative alle code di messaggi	548
11.3.2	I <i>file di lock</i>	548
11.3.3	La sincronizzazione con il <i>file locking</i>	549
11.3.4	Il <i>memory mapping</i> anonimo	551
11.4	L'intercomunicazione fra processi di POSIX	552
11.4.1	Considerazioni generali	552
11.4.2	Code di messaggi Posix	553
11.4.3	Memoria condivisa	560
11.4.4	Semafori	564
12	I thread	575
12.1	Introduzione ai <i>thread</i>	575
12.1.1	Una panoramica	575
12.1.2	<i>Thread</i> e processi	575
12.1.3	Implementazioni alternative	575
12.2	I <i>thread</i> e Linux	576
12.2.1	I <i>LinuxThread</i>	576
12.2.2	La <i>Native Thread Posix Library</i>	576
12.3	Posix <i>thread</i>	576
12.3.1	Una panoramica	576
12.3.2	La gestione dei <i>thread</i>	576
12.4	La sincronizzazione dei <i>thread</i>	576
12.4.1	I <i>mutex</i>	576
12.4.2	Le variabili di condizione	576
12.4.3	I <i>thread</i> e i segnali.	576

II	Programmazione di rete	577
13	Introduzione alla programmazione di rete	579
13.1	Modelli di programmazione	579
13.1.1	Il modello <i>client-server</i>	579
13.1.2	Il modello <i>peer-to-peer</i>	580
13.1.3	Il modello <i>three-tier</i>	580
13.1.4	Il modello <i>broadcast</i>	581
13.2	I protocolli di rete	582
13.2.1	Il modello ISO/OSI	582
13.2.2	Il modello TCP/IP (o DoD)	583
13.2.3	Criteri generali dell'architettura del TCP/IP	584
13.3	La struttura del TCP/IP	585
13.3.1	Il quadro generale	585
13.3.2	Internet Protocol (IP)	587
13.3.3	User Datagram Protocol (UDP)	588
13.3.4	Transport Control Protocol (TCP)	589
13.3.5	Limiti e dimensioni riguardanti la trasmissione dei dati	590
14	I socket	593
14.1	Introduzione ai socket	593
14.1.1	Cosa sono i <i>socket</i>	593
14.1.2	La creazione di un socket	594
14.1.3	Il dominio dei socket	595
14.1.4	Il tipo di socket	597
14.2	Le strutture degli indirizzi dei socket	598
14.2.1	La struttura generica	598
14.2.2	La struttura degli indirizzi IPv4	600
14.2.3	La struttura degli indirizzi IPv6	601
14.2.4	La struttura degli indirizzi locali	601
14.2.5	La struttura degli indirizzi AppleTalk	602
14.2.6	La struttura degli indirizzi dei <i>packet socket</i>	603
14.3	Le funzioni di conversione degli indirizzi	605
14.3.1	Le funzioni per il riordinamento	605
14.3.2	Le funzioni di conversione per gli indirizzi IPv4	605
14.3.3	Le funzioni di conversione per indirizzi IP generici	607
15	I socket TCP	609
15.1	Il funzionamento di una connessione TCP	609
15.1.1	La creazione della connessione: il <i>three way handshake</i>	609
15.1.2	Le opzioni TCP.	610
15.1.3	La terminazione della connessione	611
15.1.4	Un esempio di connessione	612
15.1.5	Lo stato <code>TIME_WAIT</code>	613
15.1.6	I numeri di porta	615
15.2	Le funzioni di base per la gestione dei socket	617
15.2.1	La funzione <code>bind</code>	618
15.2.2	La funzione <code>connect</code>	619
15.2.3	La funzione <code>listen</code>	621

15.2.4	La funzione <code>accept</code>	623
15.2.5	Le funzioni <code>getsockname</code> e <code>getpeername</code>	625
15.2.6	La funzione <code>close</code>	626
15.3	Un esempio elementare: il servizio <i>daytime</i>	626
15.3.1	Il comportamento delle funzioni di I/O	627
15.3.2	Il client <i>daytime</i>	628
15.3.3	Un server <i>daytime</i> iterativo	630
15.3.4	Un server <i>daytime</i> concorrente	632
15.4	Un esempio più completo: il servizio <i>echo</i>	635
15.4.1	Il servizio <i>echo</i>	635
15.4.2	Il client <i>echo</i> : prima versione	635
15.4.3	Il server <i>echo</i> : prima versione	637
15.4.4	L'avvio e il funzionamento ordinario	640
15.4.5	La conclusione normale	641
15.4.6	La gestione dei processi figli	642
15.5	I vari scenari critici	645
15.5.1	La terminazione precoce della connessione	646
15.5.2	La terminazione precoce del server	647
15.5.3	Altri scenari di terminazione della connessione	651
15.6	L'uso dell'I/O <i>multiplexing</i>	654
15.6.1	Il comportamento della funzione <code>select</code> con i socket.	654
15.6.2	Un esempio di I/O <i>multiplexing</i>	655
15.6.3	La funzione <code>shutdown</code>	658
15.6.4	Un server basato sull'I/O <i>multiplexing</i>	662
15.6.5	I/O <i>multiplexing</i> con <code>poll</code>	665
16	La gestione dei socket	669
16.1	La risoluzione dei nomi	669
16.1.1	La struttura del <i>resolver</i>	669
16.1.2	Le funzioni di interrogazione del DNS	671
16.1.3	La vecchia interfaccia per la risoluzione dei nomi a dominio	677
16.1.4	Le funzioni avanzate per la risoluzione dei nomi	686
16.2	Le opzioni dei socket	697
16.2.1	Le funzioni di gestione delle opzioni dei socket	698
16.2.2	Le opzioni generiche	699
16.2.3	L'uso delle principali opzioni dei socket	706
16.2.4	Le opzioni per il protocollo IPv4	713
16.2.5	Le opzioni per i protocolli TCP e UDP	719
16.3	La gestione attraverso le funzioni di controllo	726
16.3.1	L'uso di <code>ioctl</code> e <code>fcntl</code> per i socket generici	726
16.3.2	L'uso di <code>ioctl</code> per l'accesso ai dispositivi di rete	727
16.3.3	L'uso di <code>ioctl</code> per i socket TCP e UDP	731
16.4	La gestione con <code>sysctl</code> ed il filesystem <code>/proc</code>	733
16.4.1	L'uso di <code>sysctl</code> e <code>/proc</code> per le proprietà della rete	734
16.4.2	I valori di controllo per i socket generici	734
16.4.3	I valori di controllo per il protocollo IPv4	736

17 Gli altri tipi di socket	745
17.1 I socket UDP	745
17.1.1 Le caratteristiche di un socket UDP	745
17.1.2 Le funzioni <code>sendto</code> e <code>recvfrom</code>	746
17.1.3 Un client UDP elementare	749
17.1.4 Un server UDP elementare	751
17.1.5 Le problematiche dei socket UDP	753
17.1.6 L'uso della funzione <code>connect</code> con i socket UDP	757
17.2 I socket <i>Unix domain</i>	760
17.2.1 Il passaggio di file descriptor	760
17.3 Altri socket	760
17.3.1 I socket <i>raw</i>	760
17.3.2 I socket <i>netlink</i>	760
17.3.3 I <i>packet socket</i>	760
18 Socket avanzati	761
18.1 Le funzioni di I/O avanzate	761
18.1.1 La funzioni <code>send</code> e <code>recv</code>	761
18.1.2 La funzioni <code>sendmsg</code> e <code>recvmsg</code>	761
18.1.3 I messaggi ancillari	761
18.1.4 I dati <i>urgenti</i> o <i>out-of-band</i>	761
18.2 L'uso dell'I/O non bloccante	762
18.2.1 La gestione delle opzioni IP	762
 III Appendici	 763
A I codici di errore	765
A.1 Gli errori dei file	765
A.2 Gli errori dei processi	768
A.3 Gli errori di rete	768
A.4 Errori generici	770
 B Il livello di rete	 775
B.1 Il protocollo IP	775
B.1.1 Introduzione	775
B.1.2 L'intestazione di IP	777
B.1.3 Le opzioni di IP	778
B.2 Il protocollo IPv6	778
B.2.1 I motivi della transizione	779
B.2.2 Principali caratteristiche di IPv6	779
B.2.3 L'intestazione di IPv6	780
B.2.4 Gli indirizzi di IPv6	781
B.2.5 La notazione	781
B.2.6 La architettura degli indirizzi di IPv6	782
B.2.7 Indirizzi unicast <i>provider-based</i>	783
B.2.8 Indirizzi ad uso locale	784
B.2.9 Indirizzi riservati	785
B.2.10 Multicasting	785
B.2.11 Indirizzi <i>anycast</i>	786

B.2.12	Le estensioni	787
B.2.13	Qualità di servizio	788
B.2.14	Etichette di flusso	788
B.2.15	Priorità	789
B.2.16	Sicurezza a livello IP	789
B.2.17	Autenticazione	790
B.2.18	Riservatezza	791
B.2.19	Auto-configurazione	791
B.2.20	Auto-configurazione stateless	791
B.2.21	Auto-configurazione stateful	792
B.3	Il protocollo ICMP	792
B.3.1	L'intestazione di ICMP	792
C	Il livello di trasporto	795
C.1	Il protocollo TCP	795
C.1.1	Gli stati del TCP	795
C.2	Il protocollo UDP	795
D	Gli strumenti di ausilio per la programmazione	797
D.1	L'uso di make per l'automazione della compilazione	797
D.1.1	Introduzione a make	797
D.1.2	Utilizzo di make	798
D.2	Source Control Management	800
D.2.1	Introduzione a Subversion	801
D.2.2	Utilizzo di svn	802
E	Ringraziamenti	807
F	GNU Free Documentation License	809
F.1	Applicability and Definitions	809
F.2	Verbatim Copying	811
F.3	Copying in Quantity	811
F.4	Modifications	812
F.5	Combining Documents	813
F.6	Collections of Documents	813
F.7	Aggregation With Independent Works	814
F.8	Translation	814
F.9	Termination	814
F.10	Future Revisions of This License	815
F.11	Relicensing	815
F.12	Addendum: How to use this License for your documents	815

Un preambolo

Questa guida nasce dalla mia profonda convinzione che le istanze di libertà e di condivisione della conoscenza che hanno dato vita a quello straordinario movimento di persone ed intelligenza che va sotto il nome di *software libero* hanno la stessa rilevanza anche quando applicate alla produzione culturale in genere.

L'ambito più comune in cui questa filosofia viene applicata è quello della documentazione perché il software, per quanto possa essere libero, se non accompagnato da una buona documentazione che aiuti a comprenderne il funzionamento, rischia di essere fortemente deficitario riguardo ad una delle libertà fondamentali, quella di essere studiato e migliorato.

Ritengo inoltre che in campo tecnico ed educativo sia importante poter disporre di testi didattici (come manuali, enciclopedie, dizionari, ecc.) in grado di crescere, essere adattati alle diverse esigenze, modificati e ampliati, o anche ridotti per usi specifici, nello stesso modo in cui si fa per il software libero.

Questa guida è il mio tentativo di restituire indietro, nei limiti di quelle che sono le mie capacità, un po' della conoscenza che ho ricevuto, mettendo a disposizione un testo che possa fare da riferimento a chi si avvicina alla programmazione su Linux, nella speranza anche di trasmettergli non solo delle conoscenze tecniche, ma anche un po' di quella passione per la libertà e la condivisione della conoscenza che sono la ricchezza maggiore che ho ricevuto.

E, come per il software libero, anche in questo caso è importante la possibilità di accedere ai sorgenti (e non solo al risultato finale, sia questo una stampa o un file formattato) e la libertà di modificarli per apportarvi migliorie, aggiornamenti, ecc.

Per questo motivo la Free Software Foundation ha creato una apposita licenza che potesse giocare lo stesso ruolo fondamentale che la GPL ha avuto per il software libero nel garantire la permanenza delle libertà date, ma potesse anche tenere conto delle differenze che comunque ci sono fra un testo ed un programma.

Una di queste differenze è che in un testo, come in questa sezione, possono venire espresse quelle che sono le idee ed i punti di vista dell'autore, e mentre trovo che sia necessario permettere cambiamenti nei contenuti tecnici, che devono essere aggiornati e corretti, non vale lo stesso per l'espressione delle mie idee contenuta in questa sezione, che ho richiesto resti invariata.

Il progetto pertanto prevede il rilascio della guida con licenza GNU FDL, ed una modalità di realizzazione aperta che permetta di accogliere i contributi di chiunque sia interessato. Tutti i programmi di esempio sono rilasciati con licenza GNU GPL.

Prefazione

Questo progetto mira alla stesura di un testo il più completo e chiaro possibile sulla programmazione di sistema su un kernel Linux. Essendo i concetti in gran parte gli stessi, il testo dovrebbe restare valido anche per la programmazione in ambito di sistemi Unix generici, ma resta una attenzione specifica alle caratteristiche peculiari del kernel Linux e delle versioni delle librerie del C in uso con esso; in particolare si darà ampio spazio alla versione realizzata dal progetto GNU, le cosiddette *GNU C Library* o *glibc*, che ormai sono usate nella stragrande maggioranza dei casi, senza tralasciare, là dove note, le differenze con altre implementazioni come le *libc5* o le *uclib*.

L'obiettivo finale di questo progetto è quello di riuscire a ottenere un testo utilizzabile per apprendere i concetti fondamentali della programmazione di sistema della stessa qualità dei libri del compianto R. W. Stevens (è un progetto molto ambizioso ...).

Infatti benché le pagine di manuale del sistema (quelle che si accedono con il comando *man*) e il manuale delle librerie del C GNU siano una fonte inesauribile di informazioni (da cui si è costantemente attinto nella stesura di tutto il testo) la loro struttura li rende totalmente inadatti ad una trattazione che vada oltre la descrizione delle caratteristiche particolari dello specifico argomento in esame (ed in particolare lo *GNU C Library Reference Manual* non brilla per chiarezza espositiva).

Per questo motivo si è cercato di fare tesoro di quanto appreso dai testi di R. W. Stevens (in particolare [?] e [?]) per rendere la trattazione dei vari argomenti in una sequenza logica il più esplicativa possibile, corredando il tutto, quando possibile, con programmi di esempio.

Dato che sia il kernel che tutte le librerie fondamentali di GNU/Linux sono scritte in C, questo sarà il linguaggio di riferimento del testo. In particolare il compilatore usato per provare tutti i programmi e gli esempi descritti nel testo è lo GNU GCC. Il testo presuppone una conoscenza media del linguaggio, e di quanto necessario per scrivere, compilare ed eseguire un programma.

Infine, dato che lo scopo del progetto è la produzione di un libro, si è scelto di usare \LaTeX come "ambiente di sviluppo" del medesimo, sia per l'impareggiabile qualità tipografica ottenibile, che per la congruenza dello strumento con il fine, tanto sul piano pratico, quanto su quello filosofico.

Il testo sarà, almeno inizialmente, in italiano. Per il momento lo si è suddiviso in due parti, la prima sulla programmazione di sistema, in cui si trattano le varie funzionalità disponibili per i programmi che devono essere eseguiti su una singola macchina, la seconda sulla programmazione di rete, in cui si trattano le funzionalità per eseguire programmi che mettono in comunicazione macchine diverse.

Parte I

Programmazione di sistema

Capitolo 1

L'architettura del sistema

In questo primo capitolo sarà fatta un'introduzione ai concetti generali su cui è basato un sistema operativo di tipo Unix come GNU/Linux, in questo modo potremo fornire una base di comprensione mirata a sottolineare le peculiarità del sistema che sono più rilevanti per quello che riguarda la programmazione.

Dopo un'introduzione sulle caratteristiche principali di un sistema di tipo Unix passeremo ad illustrare alcuni dei concetti base dell'architettura di GNU/Linux (che sono comunque comuni a tutti i sistemi *unix-like*) ed introdurremo alcuni degli standard principali a cui viene fatto riferimento.

1.1 Una panoramica

In questa prima sezione faremo una breve panoramica sull'architettura di un sistema operativo di tipo Unix, come GNU/Linux, e della relazione fra le varie parti che lo compongono. Chi avesse già una conoscenza di questa materia può tranquillamente saltare questa sezione.

1.1.1 Concetti base

Il concetto principale su cui è basata l'architettura di un sistema unix-like è quello di un nucleo del sistema, il cosiddetto *kernel* (nel nostro caso Linux) a cui si demanda la gestione delle risorse della propria macchina (la CPU, la memoria, le periferiche) mentre tutto il resto, quindi anche la parte che prevede l'interazione con l'utente, dev'essere realizzato tramite programmi eseguiti dal kernel, che accedano alle risorse tramite opportune richieste a quest'ultimo.

Fin dai suoi albori Unix nasce come sistema operativo *multitasking*, cioè in grado di eseguire contemporaneamente più programmi, e multiutente, in cui è possibile che più utenti siano connessi ad una macchina eseguendo più programmi “*in contemporanea*”. In realtà, almeno per le macchine a processore singolo, i programmi vengono semplicemente eseguiti uno alla volta in una opportuna *rotazione*.¹

I kernel Unix più recenti, come Linux, sono realizzati sfruttando alcune caratteristiche dei processori moderni come la gestione hardware della memoria e la modalità protetta. In sostanza con i processori moderni si può disabilitare temporaneamente l'uso di certe istruzioni e l'accesso a certe zone di memoria fisica. Quello che succede è che il kernel è il solo programma

¹anche se oggi, con la presenza di sistemi multiprocessore, si possono avere più processi eseguiti in contemporanea, il concetto di “*rotazione*” resta comunque valido, dato che in genere il numero di processi da eseguire eccede il numero dei processori disponibili.

ad essere eseguito in modalità privilegiata, con il completo accesso a tutte le risorse della macchina, mentre i programmi normali vengono eseguiti in modalità protetta senza accesso diretto alle risorse. Uno schema elementare della struttura del sistema è riportato in fig. 1.1.

Figura 1.1: Schema di massima della struttura di interazione fra processi, kernel e dispositivi in Linux.

Una parte del kernel, lo *scheduler*, si occupa di stabilire, sulla base di un opportuno calcolo delle priorità e con una suddivisione appropriata del tempo di processore, quali fra i vari “processi” presenti nel sistema deve essere eseguito, realizzando il cosiddetto *preemptive multitasking*.² Ogni processo verrà comunque eseguito in modalità protetta; quando necessario esso potrà accedere alle risorse della macchina soltanto attraverso delle “chiamate al sistema” (vedi sez. 1.1.3) che restituiranno il controllo al kernel per eseguire le operazioni necessarie.

La memoria viene sempre gestita dal kernel attraverso il meccanismo della *memoria virtuale*, che consente di assegnare a ciascun processo uno spazio di indirizzi “virtuale” (vedi sez. 2.2) che il kernel stesso, con l’ausilio della unità di gestione della memoria, si incaricherà di rimappare automaticamente sulla memoria fisica disponibile, con la possibilità ulteriore di spostare temporaneamente su disco (nella cosiddetta area di *swap*) parte di detta memoria qualora ci si trovi nella necessità di liberare risorse.

Le periferiche infine vengono normalmente viste attraverso un’interfaccia astratta che permette di trattarle come se fossero dei file, secondo uno dei concetti base della architettura di Unix, per cui “*tutto è in file*” (*everything is a file*) su cui torneremo in sez. 1.2. In realtà questo non è sempre vero (ad esempio non lo è per le interfacce di rete) dato che ci sono periferiche che non rispondendo bene a questa astrazione richiedono un’interfaccia diversa. Anche in questo caso però resta valido il concetto generale che tutto il lavoro di accesso e gestione delle periferiche a basso livello viene effettuato dal kernel tramite l’opportuno codice di gestione delle stesse, che in fig. 1.1 si è indicato come *driver*.

1.1.2 Il kernel e il sistema

Uno dei concetti fondamentali su cui si basa l’architettura dei sistemi Unix è quello della distinzione fra il cosiddetto *user space*, che contraddistingue l’ambiente in cui vengono eseguiti i programmi, e il *kernel space*, che è l’ambiente in cui viene eseguito il kernel. Ogni programma vede sé stesso come se avesse la piena disponibilità della CPU e della memoria ed è, salvo i meccanismi di comunicazione previsti dal sistema, completamente ignaro del fatto che altri programmi possono essere messi in esecuzione dal kernel.

Per questa separazione non è possibile ad un singolo programma disturbare l’azione di un altro programma o del kernel stesso, e questo è il principale motivo della stabilità di un sistema unix-like nei confronti di altri sistemi in cui i processi non hanno di questi limiti o in cui essi vengono eseguiti allo stesso livello del kernel. Pertanto deve essere chiaro a chi programma in un sistema unix-like che l’accesso diretto all’hardware non può avvenire se non all’interno del kernel; al di fuori dal kernel il programmatore deve usare le opportune interfacce che quest’ultimo fornisce per i programmi in *user space*.

Per capire meglio la distinzione fra *kernel space* e *user space* si può prendere in esame la procedura di avvio di un sistema unix-like. All’accensione il *firmware* presente nella EPROM della propria macchina (per i PC compatibili il BIOS), eseguirà la procedura di avvio del

²si chiama così quella gestione del *multitasking* in cui è il kernel a decidere a chi assegnare l’uso della CPU, potendo interrompere l’esecuzione di un processo in qualunque momento.

sistema, il cosiddetto *bootstrap*,³ incaricandosi di caricare il kernel in memoria e di farne partire l'esecuzione.

A questo punto il controllo passerà al kernel, il quale però da parte sua, una volta inizializzato opportunamente l'hardware, si limiterà a due sole operazioni, montare il filesystem radice (torneremo su questo in sez. 1.2.1) e lanciare il primo processo che eseguirà il programma di inizializzazione del sistema, che in genere, visto il suo scopo, si chiama *init*.

Una volta lanciato *init* tutto il lavoro successivo verrà eseguito *user space* da questo programma, che sua volta si incaricherà di lanciare tutti gli altri programmi, fra cui ci sarà quello che si occupa di dialogare con la tastiera e lo schermo della console, quello che mette a disposizione un terminale e la *shell* da cui inviare i comandi all'utente che si vuole collegare, ed in generale tutto quanto necessario ad avere un sistema utilizzabile.

E' da rimarcare come tutto ciò che riguarda l'interazione con l'utente, che usualmente viene visto come parte del sistema, non abbia in realtà niente a che fare con il kernel, ma sia effettuato da opportuni programmi che vengono eseguiti, allo stesso modo di un qualunque programma di scrittura o di disegno e della stessa interfaccia grafica, in *user space*.

Questo significa ad esempio che il sistema di per sé non dispone di primitive per tutta una serie di operazioni (ad esempio come la copia di un file) che altri sistemi (come Windows) hanno invece al loro interno. Questo perché tutte le operazioni di normale amministrazione di un sistema, sono effettuate attraverso dei normali programmi utilizzando le interfacce di programmazione che il kernel mette a disposizione.

È per questo motivo che quando ci si riferisce al sistema nella sua interezza viene spesso sottolineato come sia corretto parlare di "GNU/Linux" e non di Linux; da solo infatti il kernel non è sufficiente, quello che costruisce un sistema operativo utilizzabile è la presenza di tutta una serie di librerie e programmi di utilità, ed i più comuni sono appunto quelli realizzati dal progetto GNU della Free Software Foundation, grazie ai quali si possono eseguire le normali operazioni che ci si aspetta da un sistema operativo.

1.1.3 *System call* e funzioni di libreria

Come illustrato in fig. 1.1 i programmi possono accedere ai servizi forniti dal kernel tramite opportune interfacce dette *system call* (*chiamate al sistema*, appunto). Si tratta di un insieme di funzioni che un programma può invocare, per le quali viene generata un'interruzione nell'esecuzione del codice del processo, passando il controllo al kernel. Sarà quest'ultimo che eseguirà in le operazioni relative alla funzione richiesta in *kernel space*, restituendo poi i risultati al chiamante.

Ogni versione di Unix ha storicamente sempre avuto un certo numero di *system call*, che sono documentate nella seconda sezione del *Manuale di programmazione di Unix*, quella cui si accede con il comando `man 2 <nome>`, ed anche Linux non fa eccezione. Queste *system call* sono poi state codificate da vari standard, che esamineremo brevemente in sez. 1.3.

Normalmente ciascuna *system call* fornita dal kernel viene associata ad una funzione con lo stesso nome definita all'interno della libreria fondamentale del sistema, quella che viene chiamata *Libreria Standard del C* (*C Standard Library*) in ragione del fatto che il primo kernel Unix e tutti i programmi eseguiti su di esso vennero scritti in questo linguaggio, usando le sue librerie. In seguito faremo riferimento alle funzioni di questa libreria che si interfacciano alle *system call* come "*funzioni di sistema*".

³il nome deriva da un'espressione gergale che significa "sollevarsi da terra tirandosi per le stringhe delle scarpe", per indicare il compito, almeno apparentemente impossibile, di far eseguire un programma a partire da un computer appena acceso che appunto non ne contiene nessuno; non è impossibile perché in realtà c'è un programma iniziale, che è il BIOS.

Questa libreria infatti, oltre alle chiamate alle *system call*, contiene anche tutta una serie di ulteriori funzioni di utilità che vengono comunemente usate nella programmazione e sono definite nei vari standard che documentano le interfacce di programmazione di un sistema unix-like. Questo concetto è importante da tener presente perché programmare in Linux significa anche essere in grado di usare le funzioni fornite dalla libreria Standard del C, in quanto né il kernel né il linguaggio C implementano direttamente operazioni ordinarie come l'allocazione dinamica della memoria, l'input/output bufferizzato sui file, la manipolazione delle stringhe, la matematica in virgola mobile, che sono comunemente usate da qualunque programma.

Tutto ciò mette nuovamente in evidenza il fatto che nella stragrande maggioranza dei casi si dovrebbe usare il nome GNU/Linux in quanto una parte essenziale del sistema, senza la quale niente funzionerebbe, è appunto la *GNU Standard C Library* (a cui faremo da qui in avanti riferimento come *glibc*), la libreria standard del C realizzata dalla Free Software Foundation nella quale sono state implementate tutte le funzioni essenziali definite negli standard POSIX e ANSI C (e molte altre), che vengono utilizzate da qualunque programma.

Si tenga comunque presente che questo non è sempre vero, dato che esistono implementazioni alternative della libreria standard del C, come la *libc5* o la *uClib*, che non derivano dal progetto GNU. La *libc5*, che era usata con le prime versioni del kernel Linux, è oggi ormai completamente soppiantata dalla *glibc*. La *uClib* invece, pur non essendo completa come la *glibc*, resta molto diffusa nel mondo dei dispositivi *embedded* per le sue dimensioni estremamente ridotte, e soprattutto per la possibilità di togliere le parti non necessarie. Pertanto costituisce un valido rimpiazzo della *glibc* in tutti quei sistemi specializzati che richiedono una minima occupazione di memoria. Infine per lo sviluppo del sistema Android è stata realizzata da Google un'altra libreria standard del C, utilizzata principalmente per evitare l'uso della *glibc*.

Tradizionalmente le funzioni specifiche della libreria standard del C sono riportate nella terza sezione del *Manuale di Programmazione di Unix* (cioè accessibili con il comando `man 3 <nome>`) e come accennato non sono direttamente associate ad una *system call* anche se, ad esempio per la gestione dei file o della allocazione dinamica della memoria, possono farne uso nella loro implementazione. Nonostante questa distinzione, fondamentale per capire il funzionamento del sistema, l'uso da parte dei programmi di una di queste funzioni resta lo stesso, sia che si tratti di una funzione interna della libreria che di una *system call*.

1.1.4 Un sistema multiutente

Linux, come gli altri kernel Unix, nasce fin dall'inizio come sistema multiutente, cioè in grado di fare lavorare più persone in contemporanea. Per questo esistono una serie di meccanismi di sicurezza, che non sono previsti in sistemi operativi monoutente, e che occorre tenere presenti. In questa sezione parleremo brevemente soltanto dei meccanismi di sicurezza tradizionali di un sistema unix-like, oggi molti di questi sono stati notevolmente estesi rispetto al modello tradizionale, ma per il momento ignoreremo queste estensioni.

Il concetto base è quello di utente (*user*) del sistema, le cui capacità sono sottoposte a limiti precisi. Sono così previsti una serie di meccanismi per identificare i singoli utenti ed una serie di permessi e protezioni per impedire che utenti diversi possano danneggiarsi a vicenda o danneggiare il sistema. Questi meccanismi sono realizzati dal kernel stesso ed attengono alle operazioni più varie, e torneremo su di essi in dettaglio più avanti.

Normalmente l'utente è identificato da un nome (il cosiddetto *username*), che ad esempio è quello che viene richiesto all'ingresso nel sistema dalla procedura di *login* (torneremo su questo in sez. 8.1.4). Questa procedura si incarica di verificare l'identità dell'utente, in genere

attraverso la richiesta di una parola d'ordine (la *password*), anche se sono possibili meccanismi diversi.⁴ Eseguita la procedura di riconoscimento in genere il sistema manda in esecuzione un programma di interfaccia (che può essere la *shell* su terminale o un'interfaccia grafica) che mette a disposizione dell'utente un meccanismo con cui questo può impartire comandi o eseguire altri programmi.

Ogni utente appartiene anche ad almeno un gruppo (il cosiddetto *default group*), ma può essere associato ad altri gruppi (i *supplementary group*), questo permette di gestire i permessi di accesso ai file e quindi anche alle periferiche, in maniera più flessibile, definendo gruppi di lavoro, di accesso a determinate risorse, ecc.

L'utente e il gruppo sono identificati dal kernel un identificativo numerico, la cui corrispondenza ad un nome espresso in caratteri è inserita nei due file */etc/passwd* e */etc/group*.⁵ Questi identificativi sono l'*user identifier*, detto in breve *user-ID*, ed indicato dall'acronimo *UID*, e il *group identifier*, detto in breve *group-ID*, ed identificato dall'acronimo *GID*, torneremo in dettaglio su questo argomento in sez. 3.2. Il kernel conosce ed utilizza soltanto questi valori numerici, i nomi ad essi associati sono interamente gestiti in *user space* con opportune funzioni di libreria, torneremo su questo argomento in sez. 6.2.1.

Grazie a questi identificativi il sistema è in grado di tenere traccia dell'utente a cui appartiene ciascun processo ed impedire ad altri utenti di interferire con quest'ultimo. Inoltre con questo sistema viene anche garantita una forma base di sicurezza interna in quanto anche l'accesso ai file (vedi sez. 4.4) è regolato da questo meccanismo di identificazione.

Infine in ogni sistema unix-like è presente uno speciale utente privilegiato, il cosiddetto *superuser*, il cui username è di norma *root*, ed il cui *UID* è zero. Esso identifica l'amministratore del sistema, che deve essere in grado di fare qualunque operazione; per l'utente *root* infatti i meccanismi di controllo cui si è accennato in precedenza sono disattivati.⁶

1.2 L'architettura di file e directory

Come accennato in sez. 1.1.1 uno dei concetti fondamentali dell'architettura di un sistema Unix è il cosiddetto *everything is a file* (*tutto è un file*), cioè il fatto che l'accesso ai vari dispositivi di input/output del computer viene effettuato attraverso un'interfaccia astratta che tratta le periferiche allo stesso modo dei normali file di dati.

In questa sezione forniremo una descrizione a grandi linee dell'architettura della gestione dei file in Linux, partendo da una introduzione ai concetti di base, per poi illustrare la struttura dell'albero dei file ed il significato dei tipi di file, concludendo con una panoramica sulle caratteristiche principali delle due interfacce con cui i processi possono effettuare l'I/O su file.

1.2.1 Una panoramica generale

Per poter accedere ai file, il kernel deve mettere a disposizione dei programmi delle opportune *system call* che consentano di leggere e scrivere il contenuto. Tutto ciò ha due aspetti: il primo è che il kernel, per il concetto dell'*everything is a file*, deve fornire una interfaccia che consenta di operare sui file, sia che questi corrispondano ai normali file di dati, o ai cosiddetti "*file*

⁴ad esempio usando la libreria PAM (*Pluggable Authentication Methods*) è possibile generalizzare i meccanismi di autenticazione e sostituire ad esempio l'uso delle password con meccanismi di identificazione biometrica; per un approfondimento dell'argomento si rimanda alla sez. 4.3.7 di [?].

⁵in realtà negli sistemi più moderni, come vedremo in sez. 6.2.1 queste informazioni possono essere mantenute, con l'uso del *Name Service Switch*, su varie tipologie di supporti, compresi server centralizzati come LDAP.

⁶i controlli infatti vengono eseguiti da uno pseudo-codice del tipo: "if (uid) { . . . }".

speciali”, come i file di dispositivo (o *device file*) che permettono di accedere alle periferiche o le *fifo* ed i socket che forniscono funzionalità di comunicazione fra processi (torneremo su questo in sez. 4.2.5).

Il secondo aspetto è che per poter utilizzare dei normali file di dati il kernel deve provvedere ad organizzare e rendere accessibile in maniera opportuna l'informazione in essi contenuta memorizzandola sullo spazio grezzo disponibile sui dischi. Questo viene fatto strutturando l'informazione sul disco attraverso quello che si chiama un “*filesystem*”. L'informazione così strutturata poi viene resa disponibile ai processi attraverso quello che viene chiamato il “*montaggio*” del filesystem nell'albero dei file, dove il contenuto sarà accessibile nella forma ordinaria di file e directory.

In Linux il concetto di *everything is a file* è stato implementato attraverso il cosiddetto *Virtual File System* (da qui in avanti VFS) che è uno strato intermedio che il kernel usa per accedere ai più svariati filesystem mantenendo la stessa interfaccia per i programmi in *user space*. Il VFS fornisce quel livello di astrazione che permette di collegare le operazioni interne del kernel per la manipolazione sui file con le *system call* relative alle operazioni di I/O, e gestisce poi l'organizzazione di dette operazioni nei vari modi in cui i diversi filesystem le effettuano, permettendo la coesistenza di filesystem differenti all'interno dello stesso albero delle directory. Approfondiremo il funzionamento di interfaccia generica fornita dal VFS in sez. 4.1.1.

In sostanza quello che accade è che quando un processo esegue una *system call* che opera su un file, il kernel chiama sempre una funzione implementata nel VFS. La funzione eseguirà le manipolazioni sulle strutture generiche e utilizzerà poi delle chiamate alle opportune funzioni del filesystem specifico a cui si fa riferimento. Saranno queste infine a chiamare le funzioni di più basso livello che eseguono le operazioni di I/O sul dispositivo fisico, secondo lo schema riportato in fig. 1.2.

Figura 1.2: Schema delle operazioni del VFS.

Questa interfaccia resta la stessa anche quando, invece che a dei normali file, si accede alle periferiche coi citati file di dispositivo, solo che in questo caso invece di usare il codice del filesystem che accede al disco, il *Virtual File System* eseguirà direttamente il codice del kernel che permette di accedere alla periferica.

Come accennato in precedenza una delle funzioni essenziali per il funzionamento dell'interfaccia dei file è quella che consente di montare un filesystem nell'albero dei file, e rendere così visibili i suoi contenuti. In un sistema unix-like infatti, a differenza di quanto avviene in altri sistemi operativi, tutti i file vengono mantenuti all'interno di un unico albero la cui radice (quella che viene chiamata *root directory*) viene montata all'avvio direttamente dal kernel.

Come accennato in sez. 1.1.2) montare la radice è, insieme al lancio di *init*,⁷ l'unica operazione che viene effettuata direttamente dal kernel in fase di avvio. Il kernel infatti, completata la fase di inizializzazione, utilizza l'indicazione passatagli dal *bootloader* su quale sia il dispositivo che contiene il filesystem da usare come punto di partenza, lo monta come radice dell'albero dei file.

Tutti gli ulteriori filesystem che possono essere disponibili su altri dispositivi dovranno a loro volta essere inseriti nell'albero, montandoli in altrettante directory del filesystem radice, su quelli che vengono chiamati *mount point*. Questo comunque avverrà sempre in un secondo

⁷l'operazione è ovviamente anche preliminare al lancio di *init*, dato il kernel deve poter accedere al file che contiene detto programma.

tempo, a cura dei programmi eseguiti nella procedura di inizializzazione del sistema, grazie alle funzioni che tratteremo in sez. 4.1.4.

1.2.2 La risoluzione del nome di file e directory

Come appena illustrato sez. 1.2.1 una delle caratteristiche distintive di un sistema unix-like è quella di avere un unico albero dei file. Un file deve essere identificato dall'utente usando quello che viene chiamato il suo *pathname*,⁸ vale a dire tramite il “percorso” (nome che talvolta viene usato come traduzione di *pathname*) che si deve fare per accedere al file a partire da una certa “*directory*”.

Una *directory* in realtà è anch'essa un file, nel senso che è anch'essa un oggetto di un filesystem, solo che è un file particolare che il kernel riconosce appositamente come tale per poterlo utilizzare come *directory*. Il suo scopo è quello di contenere una lista di nomi di file e le informazioni che associano ciascuno di questi nomi al relativo contenuto (torneremo su questo in sez. 4.1).

Dato che questi nomi possono corrispondere ad un qualunque altro oggetto del filesystem, compresa un'altra *directory*, si ottiene naturalmente un'organizzazione ad albero inserendo nomi di *directory* dentro altre *directory*. All'interno dello stesso albero si potranno poi inserire anche tutti gli altri oggetti previsti l'interfaccia del VFS (su cui torneremo in sez. 1.2.3), come le *fifo*, i collegamenti simbolici, i socket e gli stessi file di dispositivo.

La convenzione usata nei sistemi unix-like per indicare i *pathname* dei file è quella di usare il carattere “/” come separatore fra i nomi che indicano le *directory* che lo compongono. Dato che la *directory* radice sta in cima all'albero, essa viene indicata semplicemente con il *pathname* “/”.

Un file può essere indicato rispetto ad una *directory* semplicemente specificandone il nome, il manuale della *glibc* chiama i nomi contenuti nelle *directory* “componenti” (in inglese *file name components*), noi li chiameremo più semplicemente *nomi* o *voci*, riservando la parola *componenti* ai nomi che, separati da una “/”, costituiscono il *pathname*. Questi poi dovranno corrispondere, perché il *pathname* sia valido, a voci effettivamente presenti nelle *directory*, ma non è detto che un *pathname* debba per forza risultare valido.

Il procedimento con cui dato un *pathname* si individua il file a cui esso fa riferimento, è chiamato *risoluzione del percorso* (*filename resolution* o *pathname resolution*). Lo stesso procedimento ci può anche dire che il *pathname* usato non è valido. La risoluzione viene eseguita esaminando il *pathname* da sinistra a destra e localizzando ogni componente dello stesso come nome in una *directory* a partire dalla *directory* iniziale, usando il carattere “/” come separatore per scendere dall'una all'altra. Nel caso si indichi un componente vuoto il costruito “//” viene considerato equivalente a “/”.

Ovviamente perché la risoluzione abbia successo occorre che i componenti intermedi esistano e siano effettivamente *directory*, e che il file o la *directory* indicata dall'ultimo componente esista. Inoltre i permessi relativi alle *directory* indicate nel *pathname* (torneremo su questo sez. 4.4) dovranno consentire l'accesso all'intero *pathname*.

Se il *pathname* comincia con il carattere “/” la ricerca parte dalla *directory* radice del processo. Questa, a meno di non avere eseguito una *chroot* (funzione su cui torneremo in sez. 4.5.4) è la stessa per tutti i processi ed equivale alla *directory* radice dell'albero dei file montata dal kernel all'avvio del sistema; in questo caso si parla di un *pathname assoluto*.

⁸il manuale della *glibc* deprecia questa nomenclatura, che genererebbe confusione poiché *path* indica anche un insieme di *directory* su cui effettuare una ricerca (come quello in cui la shell cerca i comandi). Al suo posto viene proposto l'uso di *filename* e di componente per il nome del file all'interno della *directory*. Non seguiremo questa scelta dato che l'uso della parola *pathname* è ormai così comune che mantenerne l'uso è senz'altro più chiaro dell'alternativa proposta.

Altrimenti la ricerca parte dalla directory di lavoro corrente del processo (su cui torneremo in sez. 4.2.4) ed il *pathname* è detto *pathname relativo*.

Infine i nomi di directory “.” e “..” hanno un significato speciale e vengono inseriti in ogni directory quando questa viene creata (vedi sez. 4.2.2). Il primo fa riferimento alla directory corrente e il secondo alla directory *genitrice* (o *parent directory*) cioè la directory che contiene il riferimento alla directory corrente.

In questo modo con “..” si può usare un *pathname* relativo per indicare un file posto al di sopra della directory corrente, tornando all'indietro nell'albero dei file. Questa retromarcia però su fermerà una volta raggiunta la directory radice, perché non esistendo in questo caso una directory superiore, il nome “..” farà riferimento alla radice stessa.

1.2.3 I tipi di file

Parlare dei tipi di file su Linux, come per qualunque sistema unix-like, significa anzitutto chiarire il proprio vocabolario e sottolineare le differenze che ci sono rispetto ad altri sistemi operativi.

Come accennato in sez. 1.2.1 su Linux l'uso del *Virtual File System* consente di trattare come file oggetti molto diversi fra loro. Oltre ai normali file di dati abbiamo già accennato ad altri due di questi oggetti, i file di dispositivo e le directory, ma ne esistono altri. In genere quando si parla di tipo di file su Linux si fa riferimento a questi diversi tipi, di cui si riportato l'elenco completo in tab. 1.1.

Tipo di file		Descrizione
<i>regular file</i>	<i>file regolare</i>	Un file che contiene dei dati (l'accezione normale di file).
<i>directory</i>	<i>cartella o direttorio</i>	Un file che contiene una lista di nomi associati a degli <i>inode</i> (vedi sez. 4.1.1).
<i>symbolic link</i>	<i>collegamento simbolico</i>	Un file che contiene un riferimento ad un altro file/directory.
<i>char device</i>	<i>dispositivo a caratteri</i>	Un file <i>speciale</i> che identifica una periferica ad accesso a caratteri.
<i>block device</i>	<i>dispositivo a blocchi</i>	Un file <i>speciale</i> che identifica una periferica ad accesso a blocchi.
<i>fifo</i>	“coda”	Un file <i>speciale</i> che identifica una linea di comunicazione unidirezionale (vedi sez. 11.1.4).
<i>socket</i>	“presa”	Un file <i>speciale</i> che identifica una linea di comunicazione bidirezionale (vedi cap. 14).

Tabella 1.1: Tipologia dei file definiti nel VFS

Si tenga ben presente che questa classificazione non ha nulla a che fare con una classificazione dei file in base al loro contenuto, dato che in tal caso si avrebbe a che fare sempre e solo con dei file di dati. E non ha niente a che fare neanche con le eventuali diverse modalità con cui si possa accedere al contenuto dei file di dati. La classificazione di tab. 1.1 riguarda il tipo di oggetti gestiti dal *Virtual File System*, ed è da notare la presenza dei cosiddetti file “speciali”.

Alcuni di essi, come le *fifo* (che tratteremo in sez. 11.1.4) ed i *socket* (che tratteremo in cap. 14) non sono altro che dei riferimenti per utilizzare alcune funzionalità di comunicazione fornite dal kernel. Gli altri sono proprio quei *file di dispositivo* che costituiscono una interfaccia diretta per leggere e scrivere sui dispositivi fisici. Anche se finora li abbiamo chiamati genericamente così, essi sono tradizionalmente suddivisi in due grandi categorie, *a blocchi* e *a caratteri*, a seconda delle modalità in cui il dispositivo sottostante effettua le operazioni di I/O.

I dispositivi a blocchi (ad esempio i dischi) sono quelli corrispondono a periferiche per le quali è richiesto che l'I/O venga effettuato per blocchi di dati di dimensioni fissate (nel caso dei dischi le dimensioni di un settore), mentre i dispositivi a caratteri sono quelli per cui l'I/O può essere effettuato senza nessuna particolare struttura, ed in generale anche un byte alla volta, da cui il nome.

Una delle differenze principali con altri sistemi operativi come il VMS o Windows è che per Unix tutti i file di dati sono identici e contengono un flusso continuo di byte. Non esiste cioè differenza per come vengono visti dal sistema file di diverso contenuto o formato, come nel caso di quella fra file di testo e binari che c'è in Windows. Non c'è neanche una strutturazione a record per il cosiddetto “accesso diretto” come nel caso del VMS.⁹

Una differenza che attiene ai contenuti di un file però esiste, ed è relativa al formato dei file di testo. Nei sistemi unix-like la fine riga è codificata in maniera diversa da Windows o dal vecchio MacOS, in particolare il fine riga è il carattere LF (`\n`) al posto del CR (`\r`) del vecchio MacOS e del CR LF (`\r\n`) di Windows. Questo può causare alcuni problemi qualora nei programmi si facciano assunzioni sul terminatore della riga e per questo esistono dei programmi come `unix2dos` e `dos2unix` che effettuano una conversione fra questi due formati di testo.

Si ricordi comunque che un kernel unix-like non fornisce nessun supporto per la tipizzazione dei file di dati in base al loro contenuto e che non c'è nessun supporto per una qualche interpretazione delle estensioni (nel nome del file) da parte del kernel,¹⁰ ogni classificazione di questo tipo avviene sempre in *user space*. Gli unici file di cui il kernel deve essere in grado di capire il contenuto sono i binari dei programmi, per i quali sono supportati solo alcuni formati, anche se oggi viene usato quasi esclusivamente l'ELF.¹¹

Nonostante l'assenza di supporto da parte del kernel per la classificazione del contenuto dei file di dati, molti programmi adottano comunque delle convenzioni per i nomi dei file, ad esempio il codice C normalmente si mette in file con l'estensione “.c”. Inoltre una tecnica molto usata per classificare i contenuti da parte dei programmi è quella di utilizzare i primi byte del file per memorizzare un “*magic number*” che ne classifichi il contenuto. Il concetto è quello di un numero intero, solitamente fra 2 e 10 byte, che identifichi il contenuto seguente, dato che questi sono anche caratteri è comune trovare espresso tale numero con stringhe come “%PDF” per i PDF o “#!” per gli script. Entrambe queste tecniche, per quanto usate ed accettate in maniera diffusa, restano solo delle convenzioni il cui rispetto è demandato alle applicazioni stesse.

1.2.4 Le due interfacce per l'accesso ai file

In Linux le interfacce di programmazione per l'I/O su file due. La prima è l'interfaccia nativa del sistema, quella che il manuale della *glibc* chiama interfaccia dei “*file descriptor*” (in italiano

⁹questo vale anche per i dispositivi a blocchi: la strutturazione dell'I/O in blocchi di dimensione fissa avviene solo all'interno del kernel, ed è completamente trasparente all'utente; inoltre talvolta si parla di *accesso diretto* riferendosi alla capacità, che non ha niente a che fare con tutto ciò, di effettuare attraverso degli appositi file di dispositivo delle operazioni di I/O direttamente sui dischi senza passare attraverso un filesystem, il cosiddetto *raw access*, introdotto coi kernel della serie 2.4.x ma ormai in sostanziale disuso.

¹⁰non è così ad esempio nel filesystem HFS dei Mac, che supporta delle risorse associate ad ogni file, che specificano fra l'altro il contenuto ed il programma da usare per leggerlo; in realtà per alcuni filesystem esiste la possibilità di associare delle risorse ai file con gli *extended attributes* (vedi sez. 4.5.1), ma è una caratteristica tutt'ora poco utilizzata, dato che non corrisponde al modello classico dei file in un sistema Unix.

¹¹il nome è l'acronimo di *Executable and Linkable Format*, un formato per eseguibili binari molto flessibile ed estendibile definito nel 1995 dal *Tool Interface Standard* che per le sue caratteristiche di non essere legato a nessun tipo di processore o architettura è stato adottato da molti sistemi unix-like e non solo.

descrittori di file). Si tratta di un'interfaccia specifica dei sistemi unix-like che fornisce un accesso non bufferizzato.

L'interfaccia è essenziale, l'accesso viene detto non bufferizzato in quanto la lettura e la scrittura vengono eseguite chiamando direttamente le *system call* del kernel, anche se in realtà il kernel effettua al suo interno alcune bufferizzazioni per aumentare l'efficienza nell'accesso ai dispositivi. L'accesso viene gestito attraverso i *file descriptor* che sono rappresentati da numeri interi (cioè semplici variabili di tipo `int`). L'interfaccia è definita nell'*header file* `unistd.h` e la tratteremo in dettaglio in sez. 5.1.

La seconda interfaccia è quella che il manuale della *glibc* chiama dei *file stream* o più semplicemente degli *stream*.¹² Essa fornisce funzioni più evolute e un accesso bufferizzato, controllato dalla implementazione fatta nella *glibc*. Questa è l'interfaccia specificata dallo standard ANSI C e perciò si trova anche su tutti i sistemi non Unix. Gli *stream* sono oggetti complessi e sono rappresentati da puntatori ad una opportuna struttura definita dalle librerie del C, a cui si accede sempre in maniera indiretta utilizzando il tipo `FILE *`; l'interfaccia è definita nell'*header file* `stdio.h` e la tratteremo in dettaglio in sez. 5.3.

Entrambe le interfacce possono essere usate per l'accesso ai file come agli altri oggetti del VFS, ma per poter accedere alle operazioni di controllo (descritte in sez. 5.2.5) su un qualunque tipo di oggetto del VFS occorre usare l'interfaccia standard di Unix con i file descriptor. Allo stesso modo devono essere usati i file descriptor se si vuole ricorrere a modalità speciali di I/O come il *file locking* o l'I/O non-bloccante (vedi cap. 10).

Gli *stream* forniscono un'interfaccia di alto livello costruita sopra quella dei *file descriptor*, che permette di poter scegliere tra diversi stili di bufferizzazione. Il maggior vantaggio degli *stream* è che l'interfaccia per le operazioni di input/output è molto più ricca di quella dei *file descriptor*, che forniscono solo funzioni elementari per la lettura/scrittura diretta di blocchi di byte. In particolare gli *stream* dispongono di tutte le funzioni di formattazione per l'input e l'output adatte per manipolare anche i dati in forma di linee o singoli caratteri.

In ogni caso, dato che gli *stream* sono implementati sopra l'interfaccia standard di Unix, è sempre possibile estrarre il *file descriptor* da uno *stream* ed eseguirvi sopra operazioni di basso livello, o associare in un secondo tempo uno *stream* ad un *file descriptor* per usare l'interfaccia più sofisticata.

In generale, se non necessitano specificatamente le funzionalità di basso livello, è opportuno usare sempre gli *stream* per la loro maggiore portabilità, essendo questi ultimi definiti nello standard ANSI C; l'interfaccia con i *file descriptor* infatti segue solo lo standard POSIX.1 dei sistemi Unix, ed è pertanto di portabilità più limitata.

1.3 Gli standard

In questa sezione faremo una breve panoramica relativa ai vari standard che nel tempo sono stati formalizzati da enti, associazioni, consorzi e organizzazioni varie al riguardo ai sistemi operativi di tipo Unix o alle caratteristiche che si sono stabilite come standard di fatto in quanto facenti parte di alcune implementazioni molto diffuse come BSD o System V.

Ovviamente prenderemo in considerazione solo gli standard riguardanti interfacce di programmazione e le altre caratteristiche di un sistema unix-like (alcuni standardizzano pure i comandi base del sistema e la shell) ed in particolare ci concentreremo sul come ed in che

¹²in realtà una interfaccia con lo stesso nome è stata introdotta a livello di kernel negli Unix derivati da *System V*, come strato di astrazione per file e socket; in Linux questa interfaccia, che comunque ha avuto poco successo, non esiste, per cui facendo riferimento agli *stream* useremo il significato adottato dal manuale della *glibc*.

modo essi sono supportati sia per quanto riguarda il kernel che la libreria standard del C, con una particolare attenzione alla *glibc*.

1.3.1 Lo standard ANSI C

Lo standard ANSI C è stato definito nel 1989 dall'*American National Standard Institute* come prima standardizzazione del linguaggio C e per questo si fa riferimento ad esso anche come C89. L'anno successivo è stato adottato dalla ISO (*International Standard Organisation*) come standard internazionale con la sigla ISO/IEC 9899:1990, e per questo è noto anche sotto il nome di standard ISO C, o ISO C90. Nel 1999 è stata pubblicata una revisione dello standard C89, che viene usualmente indicata come C99, anche questa è stata ratificata dalla ISO con la sigla ISO/IEC 9899:1990, per cui vi si fa riferimento anche come ISO C99.

Scopo dello standard è quello di garantire la portabilità dei programmi C fra sistemi operativi diversi, ma oltre alla sintassi ed alla semantica del linguaggio C (operatori, parole chiave, tipi di dati) lo standard prevede anche una libreria di funzioni che devono poter essere implementate su qualunque sistema operativo.

Per questo motivo, anche se lo standard non ha alcun riferimento ad un sistema di tipo Unix, GNU/Linux (per essere precisi la *glibc*), come molti Unix moderni, provvede la compatibilità con questo standard, fornendo le funzioni di libreria da esso previste. Queste sono dichiarate in una serie di *header file* anch'essi forniti dalla *glibc* (tratteremo l'argomento in sez. 2.1.2).

In realtà la *glibc* ed i relativi *header file* definiscono un insieme di funzionalità in cui sono incluse come sottoinsieme anche quelle previste dallo standard ANSI C. È possibile ottenere una conformità stretta allo standard (scartando le funzionalità aggiuntive) usando il `gcc` con l'opzione `-ansi`. Questa opzione istruisce il compilatore a definire nei vari *header file* soltanto le funzionalità previste dallo standard ANSI C e a non usare le varie estensioni al linguaggio e al preprocessore da esso supportate.

1.3.2 I tipi di dati primitivi

Uno dei problemi di portabilità del codice più comune è quello dei tipi di dati utilizzati nei programmi, che spesso variano da sistema a sistema, o anche da una architettura ad un'altra (ad esempio passando da macchine con processori 32 bit a 64). In particolare questo è vero nell'uso dei cosiddetti *tipi elementari* del linguaggio C (come `int`) la cui dimensione varia a seconda dell'architettura hardware.

Storicamente alcuni tipi nativi dello standard ANSI C sono sempre stati associati ad alcune variabili nei sistemi Unix, dando per scontata la dimensione. Ad esempio la posizione corrente all'interno di un file è stata associata ad un intero a 32 bit, mentre il numero di dispositivo è stato associato ad un intero a 16 bit. Storicamente questi erano definiti rispettivamente come `int` e `short`, ma tutte le volte che, con l'evolversi ed il mutare delle piattaforme hardware, alcuni di questi tipi si sono rivelati inadeguati o sono cambiati, ci si è trovati di fronte ad una infinita serie di problemi di portabilità.

Per questo motivo tutte le funzioni di libreria di solito non fanno riferimento ai tipi elementari dello standard del linguaggio C, ma ad una serie di *tipi primitivi* del sistema, riportati in tab. 1.2, e definiti nell'*header file* `sys/types.h`, in modo da mantenere completamente indipendenti i tipi utilizzati dalle funzioni di sistema dai tipi elementari supportati dal compilatore C.

Tipo	Contenuto
caddr_t	Core address.
clock_t	Contatore del <i>process time</i> (vedi sez. 6.4.2).
dev_t	Numero di dispositivo (vedi sez. 4.2.5).
gid_t	Identificatore di un gruppo (vedi sez. 3.2.1).
ino_t	Numero di <i>inode</i> (vedi sez. 4.1.1).
key_t	Chiave per il System V IPC (vedi sez. 11.2.1).
loff_t	Posizione corrente in un file.
mode_t	Attributi di un file.
nlink_t	Contatore dei collegamenti su un file.
off_t	Posizione corrente in un file.
pid_t	Identificatore di un processo (vedi sez. 3.1.2).
rlim_t	Limite sulle risorse.
sigset_t	Insieme di segnali (vedi sez. 7.4.2).
size_t	Dimensione di un oggetto.
ssize_t	Dimensione in numero di byte ritornata dalle funzioni.
ptrdiff_t	Differenza fra due puntatori.
time_t	Numero di secondi (in <i>calendar time</i> , vedi sez. 6.4).
uid_t	Identificatore di un utente (vedi sez. 3.2.1).

Tabella 1.2: Elenco dei tipi primitivi, definiti in `sys/types.h`.

1.3.3 Lo standard System V

Come noto Unix nasce nei laboratori della AT&T, che ne registrò il nome come marchio depositato, sviluppandone una serie di versioni diverse; nel 1983 la versione supportata ufficialmente venne rilasciata al pubblico con il nome di Unix System V, e si fa riferimento a questa implementazione con la sigla SysV o SV.

Negli anni successivi l'AT&T proseguì lo sviluppo del sistema rilasciando varie versioni con aggiunte e integrazioni, ed in particolare la *release 2* nel 1985, a cui si fa riferimento con il nome SVr2 e la *release 3* nel 1986 (denominata SVr3). Le interfacce di programmazione di queste due versioni vennero descritte formalmente in due documenti denominati *System V Interface Definition* (o SVID), pertanto nel 1985 venne rilasciata la specifica SVID 1 e nel 1986 la specifica SVID 2.

Nel 1989 un accordo fra vari venditori (AT&T, Sun, HP, ed altri) portò ad una versione di System V che provvedeva un'unificazione delle interfacce comprendente anche Xenix e BSD, questa venne denominata *release 4* o SVr4. Anche le relative interfacce vennero descritte in un documento dal titolo *System V Interface Description*, venendo a costituire lo standard SVID 3, che viene considerato la specifica finale di System V, ed a cui spesso si fa riferimento semplicemente con SVID. Anche SVID costituisce un sovrainsieme delle interfacce definite dallo standard POSIX.

Nel 1992 venne rilasciata una seconda versione del sistema, la SVr4.2; l'anno successivo la divisione della AT&T (già a suo tempo rinominata in Unix System Laboratories) venne acquistata dalla Novell, che poi trasferì il marchio Unix al consorzio X/Open. L'ultima versione di System V fu la SVr4.2MP rilasciata nel dicembre 1993. Infine nel 1995 è stata rilasciata da SCO, che aveva acquisito alcuni diritti sul codice di System V, una ulteriore versione delle *System V Interface Description*, che va sotto la denominazione di SVID 4.

Linux e la *glibc* implementano le principali funzionalità richieste dalle specifiche SVID che non sono già incluse negli standard POSIX ed ANSI C, per compatibilità con lo Unix System V e con altri Unix (come SunOS) che le includono. Tuttavia le funzionalità più oscure e meno utilizzate (che non sono presenti neanche in System V) sono state tralasciate.

Le funzionalità implementate sono principalmente il meccanismo di intercomunicazione fra i processi e la memoria condivisa (il cosiddetto System V IPC, che vedremo in sez. 11.2)

le funzioni della famiglia `hsearch` e `drand48`, `fmtmsg` e svariate funzioni matematiche.

1.3.4 Lo “*standard*” BSD

Lo sviluppo di BSD iniziò quando la fine della collaborazione fra l'Università di Berkeley e la AT&T generò una delle prime e più importanti fratture del mondo Unix. L'università di Berkeley proseguì nello sviluppo della base di codice di cui disponeva, e che presentava parecchie migliorie rispetto alle versioni allora disponibili, fino ad arrivare al rilascio di una versione completa di Unix, chiamata appunto BSD, del tutto indipendente dal codice della AT&T.

Benché BSD non sia mai stato uno standard formalizzato, l'implementazione dello Unix dell'Università di Berkeley nella sua storia ha introdotto una serie di estensioni e interfacce di grandissima rilevanza, come i collegamenti simbolici, la funzione `select` ed i socket di rete. Per questo motivo si fa spesso riferimento esplicito alle interfacce presenti nelle varie versioni dello Unix di Berkeley con una apposita sigla.

Nel 1983, con il rilascio della versione 4.2 di BSD, venne definita una implementazione delle funzioni di interfaccia a cui si fa riferimento con la sigla 4.2BSD. Per fare riferimento alle precedenti versioni si usano poi le sigle 3BSD e 4BSD (per le due versioni pubblicate nel 1980), e 4.1BSD per quella pubblicata nel 1981.

Le varie estensioni ideate a Berkeley sono state via via aggiunte al sistema nelle varie versioni succedutesi negli anni, che vanno sotto il nome di 4.3BSD, per la versione rilasciata nel 1986 e 4.4BSD, per la versione rilasciata nel 1993, che costituisce l'ultima release ufficiale dell'università di Berkeley. Si tenga presente che molte di queste interfacce sono presenti in derivati commerciali di BSD come SunOS. Il kernel Linux e la *glibc* forniscono tutte queste estensioni che sono state in gran parte incorporate negli standard successivi.

1.3.5 Gli standard IEEE – POSIX

Lo standard ufficiale creato da un organismo indipendente più attinente alle interfacce di un sistema unix-like nel suo complesso (e che concerne sia il kernel che le librerie che i comandi) è stato lo standard POSIX. Esso prende origine dallo standard ANSI C, che contiene come sottoinsieme, prevedendo ulteriori capacità per le funzioni in esso definite, ed aggiungendone di nuove.

In realtà POSIX è una famiglia di standard diversi, il cui nome, suggerito da Richard Stallman, sta per *Portable Operating System Interface*, ma la X finale denuncia la sua stretta relazione con i sistemi Unix. Esso nasce dal lavoro dell'IEEE (*Institute of Electrical and Electronics Engineers*) che ne produsse una prima versione, nota come *IEEE 1003.1-1988*, mirante a standardizzare l'interfaccia con il sistema operativo.

Ma gli standard POSIX non si limitano alla standardizzazione delle funzioni di libreria, e in seguito sono stati prodotti anche altri standard per la shell e i comandi di sistema (1003.2), per le estensioni *real-time* e per i *thread* (rispettivamente 1003.1d e 1003.1c) per i socket (1003.1g) e vari altri. In tab. 1.3 è riportata una classificazione sommaria dei principali documenti prodotti, e di come sono identificati fra IEEE ed ISO; si tenga conto inoltre che molto spesso si usa l'estensione IEEE anche come aggiunta al nome POSIX; ad esempio è più comune parlare di POSIX.4 come di POSIX.1b.

Si tenga presente inoltre che nuove specifiche e proposte di standardizzazione si aggiungono continuamente, mentre le versioni precedenti vengono riviste; talvolta poi i riferimenti cambiano nome, per cui anche solo seguire le denominazioni usate diventa particolarmente faticoso.

Standard	IEEE	ISO	Contenuto
POSIX.1	1003.1	9945-1	Interfacce di base.
POSIX.1a	1003.1a	9945-1	Estensioni a POSIX.1.
POSIX.2	1003.2	9945-2	Comandi.
POSIX.3	2003	TR13210	Metodi di test.
POSIX.4	1003.1b	—	Estensioni real-time.
POSIX.4a	1003.1c	—	Thread.
POSIX.4b	1003.1d	9945-1	Ulteriori estensioni real-time.
POSIX.5	1003.5	14519	Interfaccia per il linguaggio ADA.
POSIX.6	1003.2c,1e	9945-2	Sicurezza.
POSIX.8	1003.1f	9945-1	Accesso ai file via rete.
POSIX.9	1003.9	—	Interfaccia per il Fortran-77.
POSIX.12	1003.1g	9945-1	Socket.

Tabella 1.3: Elenco dei vari standard POSIX e relative denominazioni.

Benché l'insieme degli standard POSIX siano basati sui sistemi Unix, essi definiscono comunque un'interfaccia di programmazione generica e non fanno riferimento ad una implementazione specifica (ad esempio esiste un'implementazione di POSIX.1 anche sotto Windows NT).

Linux e la *glibc* implementano tutte le funzioni definite nello standard POSIX.1, queste ultime forniscono in più alcune ulteriori capacità (per funzioni di *pattern matching* e per la manipolazione delle *regular expression*), che vengono usate dalla shell e dai comandi di sistema e che sono definite nello standard POSIX.2.

Nelle versioni più recenti del kernel e delle librerie sono inoltre supportate ulteriori funzionalità aggiunte dallo standard POSIX.1c per quanto riguarda i *thread* (vedi cap. 12), e dallo standard POSIX.1b per quanto riguarda i segnali e lo scheduling real-time (sez. 7.5.1 e sez. 3.3.3), la misura del tempo, i meccanismi di intercomunicazione (sez. 11.4) e l'I/O asincrono (sez. 10.3.3).

Lo standard principale resta comunque POSIX.1, che continua ad evolversi; la versione più nota, cui gran parte delle implementazioni fanno riferimento, è stata rilasciata anche come standard internazionale con la sigla *ISO/IEC 9945-1:1996* ed include i precedenti POSIX.1b e POSIX.1c. In genere si fa riferimento ad essa come POSIX.1-1996.

Nel 2001 è stata poi eseguita una sintesi degli standard POSIX.1, POSIX.2 e SUSv3 (vedi sez. 1.3.6) in un unico documento, redatto sotto gli auspici del cosiddetto gruppo Austin che va sotto il nome di POSIX.1-2001. Questo standard definisce due livelli di conformità, quello POSIX, in cui sono presenti solo le interfacce di base, e quello XSI che richiede la presenza di una serie di estensioni opzionali per lo standard POSIX, riprese da SUSv3. Inoltre lo standard è stato allineato allo standard C99, e segue lo stesso nella definizione delle interfacce.

A questo standard sono stati aggiunti due documenti di correzione e perfezionamento denominati *Technical Corrigenda*, il TC1 del 2003 ed il TC2 del 2004, e talvolta si fa riferimento agli stessi con le sigle POSIX.1-2003 e POSIX.1-2004.

Una ulteriore revisione degli standard POSIX e SUS è stata completata e ratificata nel 2008, cosa che ha portato al rilascio di una nuova versione sotto il nome di POSIX.1-2008 (e SUSv4), con l'incorporazione di alcune nuove interfacce, la obsolescenza di altre, la trasformazione da opzionali a richieste di alcune specifiche di base, oltre alle solite precisazioni ed aggiornamenti. Anche in questo caso è prevista la suddivisione in una conformità di base, e delle interfacce aggiuntive. Una ulteriore revisione è stata pubblicata nel 2017 come POSIX.1-2017.

Le procedure di aggiornamento dello standard POSIX prevedono comunque un percor-

so continuo, che prevede la possibilità di introduzione di nuove interfacce e la definizione di precisazioni ed aggiornamenti, per questo in futuro verranno rilasciate nuove versioni. Alla stesura di queste note l'ultima revisione approvata resta POSIX.1-2017, uno stato della situazione corrente del supporto degli standard è allegato alla documentazione della *glibc* e si può ottenere con il comando `man standards`.

1.3.6 Gli standard X/Open – Opengroup – Unix

Il consorzio X/Open nacque nel 1984 come consorzio di venditori di sistemi Unix per giungere ad un'armonizzazione delle varie implementazioni. Per far questo iniziò a pubblicare una serie di documentazioni e specifiche sotto il nome di *X/Open Portability Guide* a cui di norma si fa riferimento con l'abbreviazione XPG n , con n che indica la versione.

Nel 1989 il consorzio produsse una terza versione di questa guida particolarmente voluminosa (la *X/Open Portability Guide, Issue 3*), contenente una dettagliata standardizzazione dell'interfaccia di sistema di Unix, che venne presa come riferimento da vari produttori. Questo standard, detto anche XPG3 dal nome della suddetta guida, è sempre basato sullo standard POSIX.1, ma prevede una serie di funzionalità aggiuntive fra cui le specifiche delle API (sigla che sta per *Application Programmable Interface*, in italiano interfaccia di programmazione) per l'interfaccia grafica (X11).

Nel 1992 lo standard venne rivisto con una nuova versione della guida, la Issue 4, da cui la sigla XPG4, che aggiungeva l'interfaccia XTI (*X Transport Interface*) mirante a soppiantare (senza molto successo) l'interfaccia dei socket derivata da BSD. Una seconda versione della guida fu rilasciata nel 1994; questa è nota con il nome di Spec 1170 (dal numero delle interfacce, intestazioni e comandi definiti) ma si fa riferimento ad essa anche come XPG4v2.

Nel 1993 il marchio Unix passò di proprietà dalla Novell (che a sua volta lo aveva comprato dalla AT&T) al consorzio X/Open che iniziò a pubblicare le sue specifiche sotto il nome di *Single UNIX Specification* o SUS, l'ultima versione di Spec 1170 diventò così la prima versione delle *Single UNIX Specification*, detta SUS o SUSv1, ma più comunemente nota anche come *Unix 95*.

Nel 1996 la fusione del consorzio X/Open con la Open Software Foundation (nata da un gruppo di aziende concorrenti rispetto ai fondatori di X/Open) portò alla costituzione dell'*Open Group*, un consorzio internazionale che raccoglie produttori, utenti industriali, entità accademiche e governative. Attualmente il consorzio è detentore del marchio depositato Unix, e prosegue il lavoro di standardizzazione delle varie implementazioni, rilasciando periodicamente nuove specifiche e strumenti per la verifica della conformità alle stesse.

Nel 1997 fu annunciata la seconda versione delle *Single UNIX Specification*, nota con la sigla SUSv2, in questa versione le interfacce specificate salgono a 1434, e addirittura a 3030 se si considerano le stazioni di lavoro grafiche, per le quali sono inserite pure le interfacce usate da CDE che richiede sia X11 che Motif. La conformità a questa versione permette l'uso del nome *Unix 98*, usato spesso anche per riferirsi allo standard. Un altro nome alternativo di queste specifiche, date le origini, è XPG5.

Come accennato nel 2001, con il rilascio dello standard POSIX.1-2001, è stato effettuato uno sforzo di sintesi in cui sono state comprese, nella parte di interfacce estese, anche le interfacce definite nelle *Single UNIX Specification*, pertanto si può fare riferimento a detto standard, quando comprensivo del rispetto delle estensioni XSI, come SUSv3, e fregiarsi del marchio UNIX 03 se conformi ad esso.

Infine, come avvenuto per POSIX.1-2001, anche con la successiva revisione dello standard POSIX.1 (la POSIX.1-2008) è stato stabilito che la conformità completa a tutte quelle che

sono le nuove estensioni XSI previste dall'aggiornamento vada a definire la quarta versione delle *Single UNIX Specification*, chiamata appunto SUSv4.

1.3.7 Il controllo di aderenza agli standard

In Linux, se si usa la *glibc*, la conformità agli standard appena descritti può essere richiesta sia attraverso l'uso di alcune opzioni del compilatore (il `gcc`) che con la definizione di opportune costanti prima dell'inclusione dei file di intestazione (gli *header file*, vedi sez. 2.1.2) in cui le varie funzioni di libreria vengono definite.

Ad esempio se si vuole che i programmi seguano una stretta attinenza allo standard ANSI C si può usare l'opzione `-ansi` del compilatore, e non potrà essere utilizzata nessuna funzione non riconosciuta dalle specifiche standard ISO per il C. Il `gcc` possiede inoltre una specifica opzione per richiedere la conformità ad uno standard, nella forma `-std=nome`, dove `nome` può essere `c89` per indicare lo standard ANSI C (vedi sez. 1.3.1) o `c99` per indicare la conformità allo standard C99 o `c11` per indicare la conformità allo standard C11 (revisione del 2011).¹³

Per attivare le varie opzioni di controllo di aderenza agli standard è poi possibile definire delle macro di preprocessore che controllano le funzionalità che la *glibc* può mettere a disposizione:¹⁴ questo può essere fatto attraverso l'opzione `-D` del compilatore, ma è buona norma farlo inserendo gli opportuni `#define` prima della inclusione dei propri *header file* (vedi sez. 2.1.2).

Le macro disponibili per controllare l'aderenza ai vari standard messi a disposizione della *glibc*, che rendono disponibili soltanto le funzioni in essi definite, sono illustrate nel seguente elenco:

- `__STRICT_ANSI__` richiede l'aderenza stretta allo standard C ISO; viene automaticamente predefinita qualora si invochi il `gcc` con le opzioni `-ansi` o `-std=c99`.
- `_POSIX_SOURCE` definendo questa macro (considerata obsoleta) si rendono disponibili tutte le funzionalità dello standard POSIX.1 (la versione IEEE Standard 1003.1) insieme a tutte le funzionalità dello standard ISO C. Se viene anche definita con un intero positivo la macro `_POSIX_C_SOURCE` lo stato di questa non viene preso in considerazione.
- `_POSIX_C_SOURCE` definendo questa macro ad un valore intero positivo si controlla quale livello delle funzionalità specificate da POSIX viene messa a disposizione; più alto è il valore maggiori sono le funzionalità:
 - un valore uguale a "1" rende disponibili le funzionalità specificate nella edizione del 1990 (IEEE Standard 1003.1-1990);
 - valori maggiori o uguali a "2" rendono disponibili le funzionalità previste dallo standard POSIX.2 specificate nell'edizione del 1992 (IEEE Standard 1003.2-1992),
 - un valore maggiore o uguale a "199309L" rende disponibili le funzionalità previste dallo standard POSIX.1b specificate nell'edizione del 1993 (IEEE Standard 1003.1b-1993);

¹³esistono anche le possibilità di usare i valori `gnu89`, che indica l'uso delle estensioni GNU al C89, riprese poi dal C99, `gnu99` che indica il dialetto GNU del C99, o `gnu11` che indica le estensioni GNU al C11, lo standard adottato di default dipende dalla versione del `gcc`, ed all'agosto 2018 con la versione 8.2 è `gnu11`.

¹⁴le macro sono definite nel file di dichiarazione `<features.h>`, ma non è necessario includerlo nei propri programmi in quanto viene automaticamente incluso da tutti gli altri file di dichiarazione che utilizzano le macro in esso definite; si tenga conto inoltre che il file definisce anche delle ulteriori macro interne, in genere con un doppio prefisso di `_`, che non devono assolutamente mai essere usate direttamente.

- un valore maggiore o uguale a “199506L” rende disponibili le funzionalità previste dallo standard POSIX.1 specificate nell’edizione del 1996 (*ISO/IEC 9945-1:1996*), ed in particolare le definizioni dello standard POSIX.1c per i *thread*;
- a partire dalla versione 2.3.3 della *glibc* un valore maggiore o uguale a “200112L” rende disponibili le funzionalità di base previste dallo standard POSIX.1-2001, escludendo le estensioni XSI;
- a partire dalla versione 2.10 della *glibc* un valore maggiore o uguale a “200809L” rende disponibili le funzionalità di base previste dallo standard POSIX.1-2008, escludendo le estensioni XSI;
- in futuro valori superiori potranno abilitare ulteriori estensioni.

_BSD_SOURCE definendo questa macro si rendono disponibili le funzionalità derivate da BSD4.3, insieme a quelle previste dagli standard ISO C, POSIX.1 e POSIX.2; alcune delle funzionalità previste da BSD sono però in conflitto con le corrispondenti definite nello standard POSIX.1, in questo caso se la macro è definita le definizioni previste da BSD4.3 avranno la precedenza rispetto a POSIX.

A causa della natura dei conflitti con POSIX per ottenere una piena compatibilità con BSD4.3 può essere necessario anche usare una libreria di compatibilità, dato che alcune funzioni sono definite in modo diverso. In questo caso occorrerà anche usare l’opzione `-lbsd-compat` con il compilatore per indicargli di utilizzare le versioni nella libreria di compatibilità prima di quelle normali.

Si tenga inoltre presente che la preferenza verso le versioni delle funzioni usate da BSD viene mantenuta soltanto se nessuna delle ulteriori macro di specificazione di standard successivi (vale a dire una fra `_POSIX_C_SOURCE`, `_POSIX_SOURCE`, `_SVID_SOURCE`, `_XOPEN_SOURCE`, `_XOPEN_SOURCE_EXTENDED` o `_GNU_SOURCE`) è stata a sua volta attivata, nel qual caso queste hanno la precedenza. Se però si definisce `_BSD_SOURCE` dopo aver definito una di queste macro, l’effetto sarà quello di dare la precedenza alle funzioni in forma BSD. Questa macro, essendo ricompresa in `_DEFAULT_SOURCE` che è definita di default, è stata deprecata a partire dalla *glibc* 2.20.

_SVID_SOURCE definendo questa macro si rendono disponibili le funzionalità derivate da SVID. Esse comprendono anche quelle definite negli standard ISO C, POSIX.1, POSIX.2, e X/Open (XPGn) illustrati in precedenza. Questa macro, essendo ricompresa in `_DEFAULT_SOURCE` che è definita di default, è stata deprecata a partire dalla *glibc* 2.20.

_DEFAULT_SOURCE questa macro abilita le definizioni considerate il *default*, comprese quelle richieste dallo standard POSIX.1-2008, ed è sostanzialmente equivalente all’insieme di `_SVID_SOURCE`, `_BSD_SOURCE` e `_POSIX_C_SOURCE`. Essendo predefinita non è necessario usarla a meno di non aver richiesto delle definizioni più restrittive sia con altre macro che con i flag del compilatore, nel qual caso abilita le funzioni che altrimenti sarebbero disabilitate. Questa macro è stata introdotta a partire dalla *glibc* 2.19 e consente di deprecare `_SVID_SOURCE` e `_BSD_SOURCE`.

_XOPEN_SOURCE definendo questa macro si rendono disponibili le funzionalità descritte nella *X/Open Portability Guide*. Anche queste sono un sovrainsieme di quelle definite negli standard POSIX.1 e POSIX.2 ed in effetti sia **_POSIX_SOURCE** che **_POSIX_C_SOURCE** vengono automaticamente definite. Sono incluse anche ulteriori funzionalità disponibili in BSD e SVID, più una serie di estensioni a secondo dei seguenti valori:

- la definizione della macro ad un valore qualunque attiva le funzionalità specificate negli standard POSIX.1, POSIX.2 e XPG4;
- un valore di “500” o superiore rende disponibili anche le funzionalità introdotte con SUSv2, vale a dire la conformità ad Unix98;
- a partire dalla versione 2.2 della *glibc* un valore uguale a “600” o superiore rende disponibili anche le funzionalità introdotte con SUSv3, corrispondenti allo standard POSIX.1-2001 più le estensioni XSI.
- a partire dalla versione 2.10 della *glibc* un valore uguale a “700” o superiore rende disponibili anche le funzionalità introdotte con SUSv4, corrispondenti allo standard POSIX.1-2008 più le estensioni XSI.

_XOPEN_SOURCE_EXTENDED

definendo questa macro si rendono disponibili le ulteriori funzionalità necessarie la conformità al rilascio del marchio *X/Open Unix* corrispondenti allo standard Unix95, vale a dire quelle specificate da SUSv1/XPG4v2. Questa macro viene definita implicitamente tutte le volte che si imposta **_XOPEN_SOURCE** ad un valore maggiore o uguale a 500.

_ISOC99_SOURCE definendo questa macro si rendono disponibili le funzionalità previste per la revisione delle librerie standard del C introdotte con lo standard ISO C99. La macro è definita a partire dalla versione 2.1.3 della *glibc*.

Le versioni precedenti la serie 2.1.x riconoscevano le stesse estensioni con la macro **_ISOC9X_SOURCE**, dato che lo standard non era stato finalizzato, ma la *glibc* aveva già un’implementazione completa che poteva essere attivata definendo questa macro. Benché questa sia obsoleta viene tuttora riconosciuta come equivalente di **_ISOC99_SOURCE** per compatibilità.

_ISOC11_SOURCE definendo questa macro si rendono disponibili le funzionalità previste per la revisione delle librerie standard del C introdotte con lo standard ISO C11, e abilita anche quelle previste dagli standard C99 e C95. La macro è definita a partire dalla versione 2.16 della *glibc*.

_GNU_SOURCE definendo questa macro si rendono disponibili tutte le funzionalità disponibili nei vari standard oltre a varie estensioni specifiche presenti solo nella *glibc* ed in Linux. Gli standard coperti sono: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, SUS.

L’uso di **_GNU_SOURCE** è equivalente alla definizione contemporanea delle macro: **_BSD_SOURCE**, **_SVID_SOURCE**, **_POSIX_SOURCE**, **_ISOC99_SOURCE**, e inoltre di **_POSIX_C_SOURCE** con valore “200112L” (o “199506L” per le versioni della *glibc* precedenti la 2.5), **_XOPEN_SOURCE_EXTENDED** e **_XOPEN_SOURCE** con valore 600 (o 500 per le versioni della *glibc* precedenti la 2.2); oltre a queste vengono pure attivate le ulteriori due macro **_ATFILE_SOURCE**

e `_LARGEFILE64_SOURCE` che definiscono funzioni previste esclusivamente dalla *glibc*.

Benché Linux supporti in maniera estensiva gli standard più diffusi, esistono comunque delle estensioni e funzionalità specifiche, non presenti in altri standard e lo stesso vale per la *glibc*, che definisce anche delle ulteriori funzioni di libreria. Ovviamente l'uso di queste funzionalità deve essere evitato se si ha a cuore la portabilità, ma qualora questo non sia un requisito esse possono rivelarsi molto utili.

Come per l'aderenza ai vari standard, le funzionalità aggiuntive possono essere rese esplicitamente disponibili tramite la definizione di opportune macro di preprocessore, alcune di queste vengono attivate con la definizione di `_GNU_SOURCE`, mentre altre devono essere attivate esplicitamente, inoltre alcune estensioni possono essere attivate indipendentemente tramite una opportuna macro; queste estensioni sono illustrate nel seguente elenco:

`_LARGEFILE_SOURCE`

definendo questa macro si rendono disponibili alcune funzioni che consentono di superare una inconsistenza presente negli standard con i file di grandi dimensioni, ed in particolare definire le due funzioni `fseeko` e `ftello` che al contrario delle corrispettive `fseek` e `ftell` usano il tipo di dato specifico `off_t` (vedi sez. 5.3.4).

`_LARGEFILE64_SOURCE`

definendo questa macro si rendono disponibili le funzioni di una interfaccia alternativa al supporto di valori a 64 bit nelle funzioni di gestione dei file (non supportati in certi sistemi), caratterizzate dal suffisso `64` aggiunto ai vari nomi di tipi di dato e funzioni (come `off64_t` al posto di `off_t` o `lseek64` al posto di `lseek`).

Le funzioni di questa interfaccia alternativa sono state proposte come una estensione ad uso di transizione per le *Single UNIX Specification*, per consentire la gestione di file di grandi dimensioni anche nei sistemi a 32 bit, in cui la dimensione massima, espressa con un intero, non poteva superare i 2Gb. Nei nuovi programmi queste funzioni devono essere evitate, a favore dell'uso macro `_FILE_OFFSET_BITS`, che definita al valore di `64` consente di usare in maniera trasparente le funzioni dell'interfaccia classica.

`_FILE_OFFSET_BITS`

la definizione di questa macro al valore di `64` consente di attivare la conversione automatica di tutti i riferimenti a dati e funzioni a 32 bit nelle funzioni di interfaccia ai file con le equivalenti a 64 bit, senza dover utilizzare esplicitamente l'interfaccia alternativa appena illustrata. In questo modo diventa possibile usare le ordinarie funzioni per effettuare operazioni a 64 bit sui file anche su sistemi a 32 bit.¹⁵

Se la macro non è definita o è definita con valore `32` questo comportamento viene disabilitato, e sui sistemi a 32 bit verranno usate le ordinarie funzioni a 32 bit, non avendo più il supporto per file di grandi dimensioni. Su sistemi a 64 bit invece, dove il problema non sussiste, la macro non ha nessun effetto.

¹⁵basterà ricompilare il programma dopo averla definita, e saranno usate in modo trasparente le funzioni a 64 bit.

- _ATFILE_SOURCE** definendo questa macro si rendono disponibili le estensioni delle funzioni di creazione, accesso e modifica di file e directory che risolvono i problemi di sicurezza insiti nell'uso di *pathname* relativi con programmi *multi-thread* illustrate in sez. 5.2.4. Dalla *glibc* 2.10 questa macro viene definita implicitamente se si definisce **_POSIX_C_SOURCE** ad un valore maggiore o uguale di “200809L”.
- _REENTRANT** definendo questa macro, o la equivalente **_THREAD_SAFE** (fornita per compatibilità) si rendono disponibili le versioni rientranti (vedi sez. 3.4.3) di alcune funzioni, necessarie quando si usano i *thread*. Alcune di queste funzioni sono anche previste nello standard POSIX.1c, ma ve ne sono altre che sono disponibili soltanto su alcuni sistemi, o specifiche della *glibc*, e possono essere utilizzate una volta definita la macro. Oggi la macro è obsoleta, già dalla *glibc* 2.3 le funzioni erano già completamente rientranti e dalla *glibc* 2.25 la macro è equivalente ad definire **_POSIX_C_SOURCE** con un valore di “199606L” mentre se una qualunque delle altre macro che richiede un valore di conformità più alto è definita, la sua definizione non ha alcun effetto.
- _FORTIFY_SOURCE** definendo questa macro viene abilitata l'inserimento di alcuni controlli per alcune funzioni di allocazione e manipolazione di memoria e stringhe che consentono di rilevare automaticamente alcuni errori di *buffer overflow* nell'uso delle stesse. La funzionalità è stata introdotta a partire dalla versione 2.3.4 della *glibc* e richiede anche il supporto da parte del compilatore, che è disponibile solo a partire dalla versione 4.0 del gcc.
- Le funzioni di libreria che vengono messe sotto controllo quando questa funzionalità viene attivata sono, al momento della stesura di queste note, le seguenti: *memcpy*, *mempcpy*, *memmove*, *memset*, *stpcpy*, *strcpy*, *strncpy*, *strcat*, *strncat*, *sprintf*, *snprintf*, *vsprintf*, *vsnprintf*, e *gets*.
- La macro prevede due valori, con 1 vengono eseguiti dei controlli di base che non cambiano il comportamento dei programmi se si richiede una ottimizzazione di livello uno o superiore,¹⁶ mentre con il valore 2 vengono aggiunti maggiori controlli. Dato che alcuni dei controlli vengono effettuati in fase di compilazione l'uso di questa macro richiede anche la collaborazione del compilatore, disponibile dalla versione 4.0 del gcc.

Se non è stata specificata esplicitamente nessuna di queste macro il default assunto è che siano definite **_BSD_SOURCE**, **_SVID_SOURCE**, **_POSIX_SOURCE** e, con le versioni della *glibc* più recenti, che la macro **_POSIX_C_SOURCE** abbia il valore “200809L”, per versioni precedenti della *glibc* il valore assegnato a **_POSIX_C_SOURCE** era di “200112L” prima delle 2.10, di “199506L” prima delle 2.4, di “199506L” prima delle 2.1. Si ricordi infine che perché queste macro abbiano effetto devono essere sempre definite prima dell'inclusione dei file di dichiarazione.

¹⁶vale a dire se si usa l'opzione -O1 o superiore del gcc.

Capitolo 2

L'interfaccia base con i processi

Come accennato nell'introduzione il *processo* è l'unità di base con cui un sistema unix-like alloca ed utilizza le risorse. Questo capitolo tratterà l'interfaccia base fra il sistema e i processi, come vengono passati gli argomenti, come viene gestita e allocata la memoria, come un processo può richiedere servizi al sistema e cosa deve fare quando ha finito la sua esecuzione. Nella sezione finale accenneremo ad alcune problematiche generiche di programmazione.

In genere un programma viene eseguito quando un processo lo fa partire eseguendo una funzione della famiglia *exec*; torneremo su questo e sulla creazione e gestione dei processi nel prossimo capitolo. In questo affronteremo l'avvio e il funzionamento di un singolo processo partendo dal punto di vista del programma che viene messo in esecuzione.

2.1 Esecuzione e conclusione di un programma

Uno dei concetti base di Unix è che un processo esegue sempre uno ed un solo programma: si possono avere più processi che eseguono lo stesso programma ma ciascun processo vedrà la sua copia del codice (in realtà il kernel fa sì che tutte le parti uguali siano condivise), avrà un suo spazio di indirizzi, variabili proprie e sarà eseguito in maniera completamente indipendente da tutti gli altri. Questo non è del tutto vero nel caso di un programma *multi-thread*, ma la gestione dei *thread* in Linux sarà trattata a parte in cap. 12.

2.1.1 L'avvio e l'esecuzione di un programma

Quando un programma viene messo in esecuzione, cosa che può essere fatta solo con una funzione della famiglia *exec* (vedi sez. 3.1.6), il kernel esegue un opportuno codice di avvio, il cosiddetto *link-loader*, costituito dal programma *ld-linux.so*. Questo programma è una parte fondamentale del sistema il cui compito è quello della gestione delle cosiddette *librerie condivise*, quelle che nel mondo Windows sono chiamate DLL (*Dinamic Link Library*), e che invece in un sistema unix-like vengono chiamate *shared objects*.

Infatti, a meno di non aver specificato il flag *-static* durante la compilazione, tutti i programmi in Linux sono compilati facendo riferimento a librerie condivise, in modo da evitare di duplicare lo stesso codice nei relativi eseguibili e consentire un uso più efficiente della memoria, dato che il codice di uno *shared objects* viene caricato in memoria dal kernel una sola volta per tutti i programmi che lo usano.

Questo significa però che normalmente il codice di un programma è incompleto, contenendo solo i riferimenti alle funzioni di libreria che vuole utilizzare e non il relativo codice. Per

questo motivo all'avvio del programma è necessario l'intervento del *link-loader* il cui compito è caricare in memoria le librerie condivise eventualmente assenti, ed effettuare poi il collegamento dinamico del codice del programma alle funzioni di libreria da esso utilizzate prima di metterlo in esecuzione.

Il funzionamento di `ld-linux.so` è controllato da alcune variabili di ambiente e dal contenuto del file `/etc/ld.so.conf` che consentono di elencare le directory un cui cercare le librerie e determinare quali verranno utilizzate. In particolare con la variabile di ambiente `LD_LIBRARY_PATH` si possono indicare ulteriori directory rispetto a quelle di sistema in cui inserire versioni personali delle librerie che hanno la precedenza su quelle di sistema, mentre con la variabile di ambiente `LD_PRELOAD` si può passare direttamente una lista di file di librerie condivise da usare al posto di quelli di sistema. In questo modo è possibile effettuare lo sviluppo o il test di nuove librerie senza dover sostituire quelle di sistema. Ulteriori dettagli sono riportati nella pagina di manuale di `ld.so` e per un approfondimento dell'argomento si può consultare sez. 3.1.2 di [?].

Una volta completate le operazioni di inizializzazione di `ld-linux.so`, il sistema fa partire qualunque programma chiamando la funzione `main`. Sta al programmatore chiamare così la funzione principale del programma da cui si suppone che inizi l'esecuzione. In ogni caso senza questa funzione lo stesso *link-loader* darebbe luogo ad errori. Lo standard ISO C specifica che la funzione `main` può non avere argomenti o prendere due argomenti che rappresentano gli argomenti passati da linea di comando (su cui torneremo in sez. 2.3.1), in sostanza un prototipo che va sempre bene è il seguente:

```
int main (int argc, char *argv[])
{
    ...
}
```

In realtà nei sistemi Unix esiste un altro modo per definire la funzione `main`, che prevede la presenza di un terzo argomento, `char *envp[]`, che fornisce l'*ambiente* del programma; questa forma però non è prevista dallo standard POSIX.1 per cui se si vogliono scrivere programmi portabili è meglio evitarla. Per accedere all'ambiente, come vedremo in sez. 2.3.3 si usa in genere una variabile globale che viene sempre definita automaticamente.

Ogni programma viene fatto partire mettendo in esecuzione il codice contenuto nella funzione `main`, ogni altra funzione usata dal programma, che sia ottenuta da una libreria condivisa, o che sia direttamente definita nel codice, dovrà essere invocata a partire dal codice di `main`. Nel caso di funzioni definite nel programma occorre tenere conto che, nel momento stesso in cui si usano le librerie di sistema (vale a dire la *glibc*) alcuni nomi sono riservati e non possono essere utilizzati.

In particolare sono riservati a priori e non possono essere mai ridefiniti in nessun caso i nomi di tutte le funzioni, le variabili, le macro di preprocessore, ed i tipi di dati previsti dallo standard ISO C. Lo stesso varrà per tutti i nomi definiti negli *header file* che si sono esplicitamente inclusi nel programma (vedi sez. 2.1.2), ma anche se è possibile riutilizzare nomi definiti in altri *header file* la pratica è da evitare nella maniera più assoluta per non generare ambiguità.

Oltre ai nomi delle funzioni di libreria sono poi riservati in maniera generica tutti i nomi di variabili o funzioni globali che iniziano con il carattere di sottolineato ("_"), e qualunque nome che inizi con il doppio sottolineato ("__") o con il sottolineato seguito da lettera maiuscola. Questi identificativi infatti sono utilizzati per i nomi usati internamente in forma privata dalle librerie, ed evitandone l'uso si elimina il rischio di conflitti.

Infine esiste una serie di classi di nomi che sono riservati per un loro eventuale uso futuro da parte degli standard ISO C e POSIX.1, questi in teoria possono essere usati senza problemi oggi, ma potrebbero dare un conflitto con una revisione futura di questi standard, per cui è comunque opportuno evitarli, in particolare questi sono:

- i nomi che iniziano per “E” costituiti da lettere maiuscole e numeri, che potrebbero essere utilizzati per nuovi codici di errore (vedi sez. 6.5),
- i nomi che iniziano con “is” o “to” e costituiti da lettere minuscole che potrebbero essere utilizzati da nuove funzioni per il controllo e la conversione del tipo di caratteri,
- i nomi che iniziano con “LC_” e costituiti da lettere maiuscole che possono essere usati per macro attinenti alla localizzazione,
- nomi che iniziano con “SIG” o “SIG_” e costituiti da lettere maiuscole che potrebbero essere usati per nuovi nomi di segnale (vedi sez. 7.2),
- nomi che iniziano con “str”, “mem”, o “wcs” e costituiti da lettere minuscole che possono essere utilizzati per funzioni attinenti alla manipolazione delle stringhe e delle aree di memoria,
- nomi che terminano in “_t” che potrebbero essere utilizzati per la definizione di nuovi tipi di dati di sistema oltre quelli di tab. 1.2).

2.1.2 Chiamate a funzioni e *system call*

Come accennato in sez. 1.1.3 un programma può utilizzare le risorse che il sistema gli mette a disposizione attraverso l'uso delle opportune *system call*. Abbiamo inoltre appena visto come all'avvio un programma venga messo in grado di chiamare le funzioni fornite da eventuali librerie condivise da esso utilizzate.

Vedremo nel resto della guida quali sono le risorse del sistema accessibili attraverso le *system call* e tratteremo buona parte delle funzioni messe a disposizione dalla libreria standard del C, in questa sezione però si forniranno alcune indicazioni generali sul come fare perché un programma possa utilizzare queste funzioni.

In sez. 1.3 abbiamo accennato come le funzioni definite nei vari standard siano definite in una serie di *header file* (in italiano *file di intestazione*). Vengono chiamati in questo modo quei file, forniti insieme al codice delle librerie, che contengono le dichiarazioni delle variabili, dei tipi di dati, delle macro di preprocessore e soprattutto delle funzioni che fanno parte di una libreria.

Questi file sono necessari al compilatore del linguaggio C per ottenere i riferimenti ai nomi delle funzioni (e alle altre risorse) definite in una libreria, per questo quando si vogliono usare le funzioni di una libreria occorre includere nel proprio codice gli *header file* che le definiscono con la direttiva `#include`. Dato che le funzioni devono essere definite prima di poterle usare in genere gli *header file* vengono inclusi all'inizio del programma. Se inoltre si vogliono utilizzare le macro di controllo delle funzionalità fornite dai vari standard illustrate in sez. 1.3.7 queste, come accennato, dovranno a loro volta essere definite prima delle varie inclusioni.

Ogni libreria fornisce i propri file di intestazione per i quali si deve consultare la documentazione, ma in tab. 2.1 si sono riportati i principali *header file* definiti nella libreria standard del C (nel caso la *glibc*) che contengono le varie funzioni previste negli standard POSIX ed ANSI C, e che prevedono la definizione sia delle funzioni di utilità generica che delle interfacce alle *system call*. In seguito per ciascuna funzione o *system call* che tratteremo indicheremo anche quali sono gli *header file* contenenti le necessarie definizioni.

Un esempio di inclusione di questi file, preso da uno dei programmi di esempio, è il seguente, e si noti come gli *header file* possano essere referenziati con il nome fra parentesi angolari, nel

Header	Standard		Contenuto
	ANSI C	POSIX	
assert.h	●	–	Verifica le asserzioni fatte in un programma.
ctype.h	●	–	Tipi standard.
dirent.h	–	●	Manipolazione delle directory.
errno.h	–	●	Errori di sistema.
fcntl.h	–	●	Controllo sulle opzioni dei file.
limits.h	–	●	Limiti e parametri del sistema.
malloc.h	●	–	Allocazione della memoria.
setjmp.h	●	–	Salti non locali.
signal.h	–	●	Gestione dei segnali.
stdarg.h	●	–	Gestione di funzioni a argomenti variabili.
stdio.h	●	–	I/O bufferizzato in standard ANSI C.
stdlib.h	●	–	Definizioni della libreria standard.
string.h	●	–	Manipolazione delle stringhe.
time.h	–	●	Gestione dei tempi.
times.h	●	–	Gestione dei tempi.
unistd.h	–	●	Unix standard library.
utmp.h	–	●	Registro connessioni utenti.

Tabella 2.1: Elenco dei principali *header file* definiti dagli standard POSIX e ANSI C.

qual caso si indica l'uso di quelli installati con il sistema,¹ o fra virgolette, nel qual caso si fa riferimento ad una versione locale, da indicare con un *pathname* relativo:

```
#include <sys/types.h>    /* primitive system data types */
#include <dirent.h>        /* directory constants and functions */
#include <stdlib.h>         /* C standard library */
#include <unistd.h>        /* unix standard library */

#include "Gpil.h"          /* local library */
```

Si tenga presente che oltre ai nomi riservati a livello generale di cui si è parlato in sez. 2.1.1, alcuni di questi *header file* riservano degli ulteriori identificativi, il cui uso sarà da evitare, ad esempio si avrà che:

- in *dirent.h* vengono riservati i nomi che iniziano con “d_” e costituiti da lettere minuscole,
- in *fcntl.h* vengono riservati i nomi che iniziano con “l_”, “F_”, “O_” e “S_”,
- in *limits.h* vengono riservati i nomi che finiscono in “_MAX”,
- in *signal.h* vengono riservati i nomi che iniziano con “sa_” e “SA_”,
- in *sys/stat.h* vengono riservati i nomi che iniziano con “st_” e “S_”,
- in *sys/times.h* vengono riservati i nomi che iniziano con “tms_”,
- in *termios.h* vengono riservati i nomi che iniziano con “c_”, “V”, “I”, “O” e “TC” e con “B” seguito da un numero,
- in *grp.h* vengono riservati i nomi che iniziano con “gr_”,
- in *pwd.h* vengono riservati i nomi che iniziano con “pw_”,

Una volta inclusi gli *header file* necessari un programma potrà richiamare le funzioni di libreria direttamente nel proprio codice ed accedere ai servizi del kernel; come accennato infatti normalmente ogni *system call* è associata ad una omonima funzione di libreria, che è quella che si usa normalmente per invocarla.

¹in un sistema GNU/Linux che segue le specifiche del *Filesystem Hierarchy Standard* (per maggiori informazioni si consulti sez. 1.2.3 di [?]) si trovano sotto */usr/include*.

Occorre però tenere presente che anche se dal punto di vista della scrittura del codice la chiamata di una *system call* non è diversa da quella di una qualunque funzione ordinaria, la situazione è totalmente diversa nell'esecuzione del programma. Una funzione ordinaria infatti viene eseguita, esattamente come il codice che si è scritto nel corpo del programma, in *user space*. Quando invece si esegue una *system call* l'esecuzione ordinaria del programma viene interrotta con quello che viene usualmente chiamato un *context switch*;² il contesto di esecuzione del processo viene salvato in modo da poterne riprendere in seguito l'esecuzione ed i dati forniti (come argomenti della chiamata) vengono trasferiti al kernel che esegue il codice della *system call* (che è codice del kernel) in *kernel space*; al completamento della *system call* i dati salvati nel *context switch* saranno usati per riprendere l'esecuzione ordinaria del programma.

Dato che il passaggio dei dati ed il salvataggio del contesto di esecuzione sono operazioni critiche per le prestazioni del sistema, per rendere il più veloce possibile questa operazione sono state sviluppate una serie di ottimizzazioni che richiedono alcune preparazioni abbastanza complesse dei dati, che in genere dipendono dall'architettura del processore e sono scritte direttamente in *assembler*.

Inoltre alcune *system call* sono state modificate nel corso degli anni con lo sviluppo del kernel per aggiungere ad esempio funzionalità in forma di nuovi argomenti, o per consolidare diverse varianti in una interfaccia generica. Per questo motivo dovendo utilizzare una *system call* è sempre preferibile usare l'interfaccia fornita dalla *glibc*, che si cura di mantenere una uniformità chiamando le versioni più aggiornate.

Ci sono alcuni casi però in cui può essere necessario evitare questa associazione, e lavorare a basso livello con una specifica versione, oppure si può voler utilizzare una *system call* che non è stata ancora associata ad una funzione di libreria. In tal caso, per evitare di dover effettuare esplicitamente le operazioni di preparazione citate, all'interno della *glibc* è fornita una specifica funzione, `syscall`,³ che consente eseguire direttamente una *system call*; il suo prototipo, accessibile se si è definita la macro `_GNU_SOURCE`, è:

```
#include <unistd.h>
#include <sys/syscall.h>
long syscall(int number, ...)
```

Esegue la *system call* indicata da `number`.

La funzione ritorna un intero dipendente dalla *system call* invocata, in generale 0 indica il successo ed un valore negativo un errore.

La funzione richiede come primo argomento il numero della *system call* da invocare, seguita dagli argomenti da passare alla stessa, che ovviamente dipendono da quest'ultima, e restituisce il codice di ritorno della *system call* invocata. In generale un valore nullo indica il successo ed un valore negativo è un codice di errore che poi viene memorizzato nella variabile `errno` (sulla gestione degli errori torneremo in dettaglio in sez. 6.5).

Il valore di `number` dipende sia dalla versione di kernel che dall'architettura,⁴ ma ciascuna *system call* viene in genere identificata da una costante nella forma `SYS_*` dove al prefisso viene aggiunto il nome che spesso corrisponde anche alla omonima funzione di libreria. Queste

²in realtà si parla più comunemente di *context switch* quando l'esecuzione di un processo viene interrotta dal kernel (tramite lo *scheduler*) per metterne in esecuzione un altro, ma il concetto generale resta lo stesso: l'esecuzione del proprio codice in *user space* viene interrotta e lo stato del processo deve essere salvato per poterne riprendere l'esecuzione in un secondo tempo.

³fino a prima del kernel 2.6.18 per l'esecuzione diretta delle *system call* erano disponibili anche una serie di macro `_syscallN` (con *N* pari al numero di argomenti della *system call*); queste sono deprecate e pertanto non ne parleremo ulteriormente.

⁴in genere le vecchie *system call* non vengono eliminate e se ne aggiungono di nuove con nuovi numeri.

costanti sono definite nel file `sys/syscall.h`, ma si possono anche usare direttamente valori numerici.

2.1.3 La terminazione di un programma

Normalmente un programma conclude la sua esecuzione quando si fa ritornare la funzione `main`, si usa cioè l'istruzione `return` del linguaggio C all'interno della stessa, o se si richiede esplicitamente la chiusura invocando direttamente la funzione `exit`. Queste due modalità sono assolutamente equivalenti, dato che `exit` viene chiamata in maniera trasparente anche quando `main` ritorna, passandogli come argomento il valore indicato da `return`.

La funzione `exit`, che è completamente generale, essendo definita dallo standard ANSI C, è quella che deve essere invocata per una terminazione “normale”, il suo prototipo è:

```
#include <unistd.h>
void exit(int status)
```

Causa la conclusione ordinaria del programma.

La funzione non ritorna, il processo viene terminato.

La funzione è pensata per eseguire una conclusione pulita di un programma che usi la libreria standard del C; essa esegue tutte le funzioni che sono state registrate con `atexit` e `on_exit` (vedi sez. 2.1.4), chiude tutti gli *stream* (vedi sez. 5.3.1) effettuando il salvataggio dei dati sospesi (chiamando `fclose`, vedi sez. 5.3.3), infine passa il controllo al kernel chiamando la *system call* `_exit` (che vedremo a breve) che completa la terminazione del processo.

Il valore dell'argomento `status` o il valore di ritorno di `main` costituisce quello che viene chiamato lo *stato di uscita* (l'*exit status*) del processo. In generale si usa questo valore per fornire al processo padre (come vedremo in sez. 3.1.5) delle informazioni generiche sulla riuscita o il fallimento del programma appena terminato.

Anche se l'argomento `status` (ed il valore di ritorno di `main`) sono numeri interi di tipo `int`, si deve tener presente che il valore dello stato di uscita viene comunque troncato ad 8 bit, per cui deve essere sempre compreso fra 0 e 255. Si tenga presente che se si raggiunge la fine della funzione `main` senza ritornare esplicitamente si ha un valore di uscita indefinito, è pertanto consigliabile di concludere sempre in maniera esplicita detta funzione.

Non esiste un significato intrinseco dello stato di uscita, ma una convenzione in uso pressoché universale è quella di restituire 0 in caso di successo e 1 in caso di fallimento. Una eccezione a questa convenzione è per i programmi che effettuano dei confronti (come `diff`), che usano 0 per indicare la corrispondenza, 1 per indicare la non corrispondenza e 2 per indicare l'incapacità di effettuare il confronto. Un'altra convenzione riserva i valori da 128 a 256 per usi speciali: ad esempio 128 viene usato per indicare l'incapacità di eseguire un altro programma in un sottoprocesso. Benché le convenzioni citate non siano seguite universalmente è una buona idea tenerle presenti ed adottarle a seconda dei casi.

Si tenga presente inoltre che non è una buona idea usare eventuali codici di errore restituiti nella variabile `errno` (vedi sez. 6.5) come *exit status*. In generale infatti non ci si cura del valore dello stato di uscita di un processo se non per vedere se è diverso da zero, come indicazione di un qualche errore. Dato che viene troncato ad 8 bit utilizzare un intero di valore generico può comportare il rischio, qualora si vada ad usare un multiplo di 256, di avere uno stato di uscita uguale a zero, che verrebbe interpretato come un successo.

Per questo motivo in `stdlib.h` sono definite, seguendo lo standard POSIX, le due costanti `EXIT_SUCCESS` e `EXIT_FAILURE`, da usare sempre per specificare lo stato di uscita di un processo. Su Linux, ed in generale in qualunque sistema POSIX, ad esse sono assegnati rispettivamente i valori 0 e 1.

Una forma alternativa per effettuare una terminazione esplicita di un programma è quella di chiamare direttamente la *system call* `_exit`,⁵ che restituisce il controllo direttamente al kernel, concludendo immediatamente il processo, il suo prototipo è:

```
#include <unistd.h>
void _exit(int status)
```

Causa la conclusione immediata del programma.

La funzione non ritorna, il processo viene terminato.

La funzione termina immediatamente il processo e le eventuali funzioni registrate con `atexit` e `on_exit` non vengono eseguite. La funzione chiude tutti i file descriptor appartenenti al processo, cosa che però non comporta il salvataggio dei dati eventualmente presenti nei buffer degli *stream*, (torneremo sulle due interfacce dei file in sez. 5.1 e sez. 5.3). Infine fa sì che ogni figlio del processo sia adottato da `init` (vedi sez. 3.1.4), manda un segnale `SIGCHLD` al processo padre (vedi sez. 7.2.6) e salva lo stato di uscita specificato in `status` che può essere raccolto usando la funzione `wait` (vedi sez. 3.1.5).

Si tenga presente infine che oltre alla conclusione “normale” appena illustrata esiste anche la possibilità di una conclusione “anomala” del programma a causa della ricezione di un segnale (tratteremo i segnali in cap. 7) o della chiamata alla funzione `abort`; torneremo su questo in sez. 3.1.4.

2.1.4 Esecuzione di funzioni preliminari all'uscita

Un'esigenza comune che si incontra è quella di dover effettuare una serie di operazioni di pulizia prima della conclusione di un programma, ad esempio salvare dei dati, ripristinare delle impostazioni, eliminare dei file temporanei, ecc. In genere queste operazioni vengono fatte in un'apposita sezione del programma, ma quando si realizza una libreria diventa antipatico dover richiedere una chiamata esplicita ad una funzione di pulizia al programmatore che la utilizza.

È invece molto meno soggetto ad errori, e completamente trasparente all'utente, avere la possibilità di fare effettuare automaticamente la chiamata ad una funzione che effettui tali operazioni all'uscita dal programma. A questo scopo lo standard ANSI C prevede la possibilità di registrare un certo numero di funzioni che verranno eseguite all'uscita dal programma,⁶ sia per la chiamata ad `exit` che per il ritorno di `main`. La prima funzione che si può utilizzare a tal fine è `atexit`, il cui prototipo è:

```
#include <stdlib.h>
int atexit(void (*function)(void))
```

Registra la funzione `function` per la chiamata all'uscita dal programma.

La funzione ritorna 0 in caso di successo e -1 per un errore, `errno` non viene modificata.

La funzione richiede come argomento `function` l'indirizzo di una opportuna funzione di pulizia da chiamare all'uscita del programma, che non deve prendere argomenti e non deve ritornare niente. In sostanza deve la funzione di pulizia dovrà essere definita come `void function(void)`.

⁵la stessa è definita anche come `_Exit` in `stdlib.h`, inoltre a partire dalla *glibc* 2.3 usando questa funzione viene invocata `exit_group` che termina tutti i *thread* del processo e non solo quello corrente (fintanto che non si usano i *thread*, vedi sez. 12, questo non fa nessuna differenza).

⁶nel caso di `atexit` lo standard POSIX.1-2001 richiede che siano registrabili almeno `ATEXIT_MAX` funzioni (il valore può essere ottenuto con `sysconf`, vedi sez. 6.1.1).

Un'estensione di `atexit` è la funzione `on_exit`, che la *glibc* include per compatibilità con SunOS ma che non è detto sia definita su altri sistemi,⁷ il suo prototipo è:

```
#include <stdlib.h>
int on_exit(void (*function)(int, void *), void *arg))
    Registra la funzione function per la chiamata all'uscita dal programma.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, `errno` non viene modificata.

In questo caso la funzione da chiamare all'uscita prende i due argomenti specificati nel prototipo, un intero ed un puntatore; dovrà cioè essere definita come `void function(int status, void *argp)`. Il primo argomento sarà inizializzato allo stato di uscita con cui è stata chiamata `exit` ed il secondo al puntatore `arg` passato come secondo argomento di `on_exit`. Così diventa possibile passare dei dati alla funzione di chiusura.

Nella sequenza di chiusura tutte le funzioni registrate verranno chiamate in ordine inverso rispetto a quello di registrazione, ed una stessa funzione registrata più volte sarà chiamata più volte. Siccome entrambe le funzioni `atexit` e `on_exit` fanno riferimento alla stessa lista, l'ordine di esecuzione sarà riferito alla registrazione in quanto tale, indipendentemente dalla funzione usata per farla.

Una volta completata l'esecuzione di tutte le funzioni registrate verranno chiusi tutti gli *stream* aperti ed infine verrà chiamata `_exit` per la terminazione del programma. Questa è la sequenza ordinaria, eseguita a meno che una delle funzioni registrate non esegua al suo interno `_exit`, nel qual caso la terminazione del programma sarà immediata ed anche le successive funzioni registrate non saranno invocate.

Se invece all'interno di una delle funzioni registrate si chiama un'altra volta `exit` lo standard POSIX.1-2001 prescrive un comportamento indefinito, con la possibilità (che su Linux comunque non c'è) di una ripetizione infinita. Pertanto questa eventualità è da evitare nel modo più assoluto. Un altro comportamento indefinito si può avere se si termina l'esecuzione di una delle funzioni registrate con `longjmp` (vedi sez. 2.4.3).

Si tenga presente infine che in caso di terminazione anomala di un processo (ad esempio a causa di un segnale) nessuna delle funzioni registrate verrà eseguita e che se invece si crea un nuovo processo con `fork` (vedi sez. 3.1.3) questo manterrà tutte le funzioni già registrate.

2.1.5 Un riepilogo

Data l'importanza dell'argomento è opportuno un piccolo riepilogo dei fatti essenziali relativi alla esecuzione di un programma. Il primo punto da sottolineare è che in un sistema unix-like l'unico modo in cui un programma può essere eseguito dal kernel è attraverso la chiamata alla *system call* `execve`, sia direttamente che attraverso una delle funzioni della famiglia `exec` che ne semplificano l'uso (vedi sez. 3.1.6).

Allo stesso modo l'unico modo in cui un programma può concludere volontariamente la propria esecuzione è attraverso una chiamata alla *system call* `_exit`, sia che questa venga fatta esplicitamente, o in maniera indiretta attraverso l'uso di `exit` o il ritorno di `main`.

Uno schema riassuntivo che illustra le modalità con cui si avvia e conclude normalmente un programma è riportato in fig. 2.1.

Figura 2.1: Schema dell'avvio e della conclusione di un programma.

⁷la funzione è disponibile dalla *glibc* 2.19 definendo la macro `_DEFAULT_SOURCE`, mentre in precedenza erano necessarie `_BSD_SOURCE` o `_SVID_SOURCE`; non essendo prevista dallo standard POSIX è in generale preferibile evitarne l'uso.

Si ricordi infine che un programma può anche essere interrotto dall'esterno attraverso l'uso di un segnale (modalità di conclusione non mostrata in fig. 2.1); tratteremo nei dettagli i segnali e la loro gestione nel capitolo 7.

2.2 I processi e l'uso della memoria

Una delle risorse più importanti che ciascun processo ha a disposizione è la memoria, e la gestione della memoria è appunto uno degli aspetti più complessi di un sistema unix-like. In questa sezione, dopo una breve introduzione ai concetti di base, esamineremo come la memoria viene vista da parte di un programma in esecuzione, e le varie funzioni utilizzabili per la sua gestione.

2.2.1 I concetti generali

Ci sono vari modi in cui i sistemi operativi organizzano la memoria, ed i dettagli di basso livello dipendono spesso in maniera diretta dall'architettura dell'hardware, ma quello più tipico, usato dai sistemi unix-like come Linux è la cosiddetta *memoria virtuale* che consiste nell'assegnare ad ogni processo uno spazio virtuale di indirizzamento lineare, in cui gli indirizzi vanno da zero ad un qualche valore massimo.⁸ Come accennato nel cap. 1 questo spazio di indirizzi è virtuale e non corrisponde all'effettiva posizione dei dati nella RAM del computer. In generale detto spazio non è neppure continuo, cioè non tutti gli indirizzi possibili sono utilizzabili, e quelli usabili non sono necessariamente adiacenti.

Per la gestione da parte del kernel la memoria viene divisa in pagine di dimensione fissa. Inizialmente queste pagine erano di 4kb sulle macchine a 32 bit e di 8kb sulle alpha. Con le versioni più recenti del kernel è possibile anche utilizzare pagine di dimensioni maggiori (di 4Mb, dette *huge page*), per sistemi con grandi quantitativi di memoria in cui l'uso di pagine troppo piccole comporta una perdita di prestazioni. In alcuni sistemi la costante `PAGE_SIZE`, definita in `limits.h`, indica la dimensione di una pagina in byte, con Linux questo non avviene e per ottenere questa dimensione si deve ricorrere alla funzione `getpagesize` (vedi sez. 6.3.3).

Ciascuna pagina di memoria nello spazio di indirizzi virtuale è associata ad un supporto che può essere una pagina di memoria reale o ad un dispositivo di stoccaggio secondario (come lo spazio disco riservato alla *swap*, o i file che contengono il codice). Per ciascun processo il kernel si cura di mantenere un mappa di queste corrispondenze nella cosiddetta *page table*.⁹

Una stessa pagina di memoria reale può fare da supporto a diverse pagine di memoria virtuale appartenenti a processi diversi, come accade in genere per le pagine che contengono il codice delle librerie condivise. Ad esempio il codice della funzione `printf` starà su una sola pagina di memoria reale che farà da supporto a tutte le pagine di memoria virtuale di tutti i processi che hanno detta funzione nel loro codice.

La corrispondenza fra le pagine della memoria virtuale di un processo e quelle della memoria fisica della macchina viene gestita in maniera trasparente dal kernel.¹⁰ Poiché in genere la memoria fisica è solo una piccola frazione della memoria virtuale, è necessario un meccanismo che permetta di trasferire le pagine che servono dal supporto su cui si trovano in memoria,

⁸nel caso di Linux fino al kernel 2.2 detto massimo era, per macchine a 32bit, di 2Gb. Con il kernel 2.4 ed il supporto per la *high-memory* il limite è stato esteso anche per macchine a 32 bit.

⁹questa è una semplificazione brutale, il meccanismo è molto più complesso; una buona trattazione di come Linux gestisce la memoria virtuale si trova su [?].

¹⁰in genere con l'ausilio dell'hardware di gestione della memoria (la *Memory Management Unit* del processore), con i kernel della serie 2.6 è comunque diventato possibile utilizzare Linux anche su architetture che non dispongono di una MMU.

eliminando quelle che non servono. Questo meccanismo è detto *paginazione* (o *paging*), ed è uno dei compiti principali del kernel.

Quando un processo cerca di accedere ad una pagina che non è nella memoria reale, avviene quello che viene chiamato un *page fault*; la gestione della memoria genera un'interruzione e passa il controllo al kernel il quale sospende il processo e si incarica di mettere in RAM la pagina richiesta, effettuando tutte le operazioni necessarie per reperire lo spazio necessario, per poi restituire il controllo al processo.

Dal punto di vista di un processo questo meccanismo è completamente trasparente, e tutto avviene come se tutte le pagine fossero sempre disponibili in memoria. L'unica differenza avvertibile è quella dei tempi di esecuzione, che passano dai pochi nanosecondi necessari per l'accesso in RAM se la pagina è direttamente disponibile, a tempi estremamente più lunghi, dovuti all'intervento del kernel, qualora sia necessario reperire pagine riposte nella *swap*.

Normalmente questo è il prezzo da pagare per avere un *multitasking* reale, ed in genere il sistema è molto efficiente in questo lavoro; quando però ci siano esigenze specifiche di prestazioni è possibile usare delle funzioni che permettono di bloccare il meccanismo della paginazione e mantenere fisse delle pagine in memoria (vedi sez. 2.2.4).

2.2.2 La struttura della memoria di un processo

Benché lo spazio di indirizzi virtuali copra un intervallo molto ampio, solo una parte di essi è effettivamente allocato ed utilizzabile dal processo; il tentativo di accedere ad un indirizzo non allocato è un tipico errore che si commette quando si è manipolato male un puntatore e genera quella che viene chiamata una *segment violation*. Se si tenta cioè di leggere o scrivere con un indirizzo per il quale non esiste un'associazione nella memoria virtuale, il kernel risponde al relativo *page fault* mandando un segnale SIGSEGV al processo, che normalmente ne causa la terminazione immediata.

È pertanto importante capire come viene strutturata la memoria virtuale di un processo. Essa viene divisa in *segmenti*, cioè un insieme contiguo di indirizzi virtuali ai quali il processo può accedere. Solitamente un programma C viene suddiviso nei seguenti segmenti:

1. Il *segmento di testo* o *text segment*. Contiene il codice del programma, delle funzioni di librerie da esso utilizzate, e le costanti. Normalmente viene condiviso fra tutti i processi che eseguono lo stesso programma e nel caso delle librerie anche da processi che eseguono altri programmi.

Quando l'architettura hardware lo supporta viene marcato in sola lettura per evitare sovrascritture accidentali (o maliziose) che ne modifichino le istruzioni. Viene allocato da *execve* all'avvio del programma e resta invariato per tutto il tempo dell'esecuzione.

2. Il *segmento dei dati* o *data segment*. Contiene tutti i dati del programma, come le *variabili globali*, cioè quelle definite al di fuori di tutte le funzioni che compongono il programma, e le *variabili statiche*, cioè quelle dichiarate con l'attributo *static*,¹¹ e la memoria allocata dinamicamente. Di norma è diviso in tre parti:

- Il segmento dei dati inizializzati, che contiene le variabili il cui valore è stato assegnato esplicitamente. Ad esempio se si definisce:

¹¹la direttiva *static* indica al compilatore C che una variabile così dichiarata all'interno di una funzione deve essere mantenuta staticamente in memoria (nel segmento dati appunto); questo significa che la variabile verrà inizializzata una sola volta alla prima invocazione della funzione e che il suo valore sarà mantenuto fra diverse esecuzioni della funzione stessa, la differenza con una variabile globale è che essa può essere vista solo all'interno della funzione in cui è dichiarata.


```
double pi = 3.14;
```

questo valore sarà immagazzinato in questo segmento. La memoria di questo segmento viene preallocata all'avvio del programma e inizializzata ai valori specificati.

- Il segmento dei dati non inizializzati, che contiene le variabili il cui valore non è stato assegnato esplicitamente. Ad esempio se si definisce:

```
int vect[100];
```

questo vettore sarà immagazzinato in questo segmento. Anch'esso viene allocato all'avvio, e tutte le variabili vengono inizializzate a zero (ed i puntatori a NULL).¹² Storicamente questa seconda parte del segmento dati viene chiamata BSS (da *Block Started by Symbol*). La sua dimensione è fissa.

- Lo *heap*, detto anche *free store*. Tecnicamente lo si può considerare l'estensione del segmento dei dati non inizializzati, a cui di solito è posto giusto di seguito. Questo è il segmento che viene utilizzato per l'allocazione dinamica della memoria. Lo *heap* può essere ridimensionato allargandolo e restringendolo per allocare e deallocare la memoria dinamica con le apposite funzioni (vedi sez. 2.2.3), ma il suo limite inferiore, quello adiacente al segmento dei dati non inizializzati, ha una posizione fissa.
3. Il segmento di *stack*, che contiene quello che viene chiamato lo “*stack*” del programma. Tutte le volte che si effettua una chiamata ad una funzione è qui che viene salvato l'indirizzo di ritorno e le informazioni dello stato del chiamante (come il contenuto di alcuni registri della CPU), poi la funzione chiamata alloca qui lo spazio per le sue variabili locali. Tutti questi dati vengono *impilati* (da questo viene il nome *stack*) in sequenza uno sull'altro; in questo modo le funzioni possono essere chiamate ricorsivamente. Al ritorno della funzione lo spazio è automaticamente rilasciato e “*ripulito*”.¹³

La dimensione di questo segmento aumenta seguendo la crescita dello *stack* del programma, ma non viene ridotta quando quest'ultimo si restringe.

Figura 2.2: Disposizione tipica dei segmenti di memoria di un processo.

Una disposizione tipica dei vari segmenti (testo, dati inizializzati e non inizializzati, *heap*, *stack*, ecc.) è riportata in fig. 2.2. Si noti come in figura sia indicata una ulteriore regione, marcata *environment*, che è quella che contiene i dati relativi alle variabili di ambiente passate al programma al suo avvio (torneremo su questo argomento in sez. 2.3.3).

Usando il comando `size` su un programma se ne può stampare le dimensioni dei segmenti di testo e di dati (solo però per i dati inizializzati ed il BSS, dato che lo *heap* ha una dimensione dinamica). Si tenga presente comunque che il BSS, contrariamente al segmento dei dati inizializzati, non è mai salvato sul file che contiene l'eseguibile, dato che viene sempre inizializzato a zero al caricamento del programma.

¹²si ricordi che questo vale solo per le variabili che vanno nel segmento dati, e non è affatto vero in generale.

¹³il compilatore si incarica di generare automaticamente il codice necessario, seguendo quella che viene chiamata una *calling convention*; quella standard usata con il C ed il C++ è detta *cdecl* e prevede che gli argomenti siano caricati nello *stack* dal chiamante da destra a sinistra, e che sia il chiamante stesso ad eseguire la ripulitura dello *stack* al ritorno della funzione, se ne possono però utilizzare di alternative (ad esempio nel Pascal gli argomenti sono inseriti da sinistra a destra ed è compito del chiamato ripulire lo *stack*), in genere non ci si deve preoccupare di questo fintanto che non si mescolano funzioni scritte con linguaggi diversi.

2.2.3 Allocazione della memoria per i programmi C

Il C supporta direttamente, come linguaggio di programmazione, soltanto due modalità di allocazione della memoria: l'*allocazione statica* e l'*allocazione automatica*.

L'*allocazione statica* è quella con cui sono memorizzate le variabili globali e le variabili statiche, cioè le variabili il cui valore deve essere mantenuto per tutta la durata del programma. Come accennato queste variabili vengono allocate nel segmento dei dati all'avvio del programma come parte delle operazioni svolte da `exec`, e lo spazio da loro occupato non viene liberato fino alla sua conclusione.

L'*allocazione automatica* è quella che avviene per gli argomenti di una funzione e per le sue variabili locali, quelle che vengono definite all'interno della funzione che esistono solo per la durata della sua esecuzione e che per questo vengono anche dette *variabili automatiche*. Lo spazio per queste variabili viene allocato nello *stack* quando viene eseguita la funzione e liberato quando si esce dalla medesima.

Esiste però un terzo tipo di allocazione, l'*allocazione dinamica* della memoria, che non è prevista direttamente all'interno del linguaggio C, ma che è necessaria quando il quantitativo di memoria che serve è determinabile solo durante il corso dell'esecuzione del programma. Il C non consente di usare variabili allocate dinamicamente, non è possibile cioè definire in fase di programmazione una variabile le cui dimensioni possano essere modificate durante l'esecuzione del programma. Per questo la libreria standard del C fornisce una opportuna serie di funzioni per eseguire l'allocazione dinamica di memoria, che come accennato avviene nello *heap*.

Le variabili il cui contenuto è allocato in questo modo non potranno essere usate direttamente come le altre (quelle nello *stack*), ma l'accesso sarà possibile solo in maniera indiretta, attraverso i puntatori alla memoria loro riservata che si sono ottenuti dalle funzioni di allocazione.

Le funzioni previste dallo standard ANSI C per la gestione della memoria sono quattro: `malloc`, `calloc`, `realloc` e `free`. Le prime due, `malloc` e `calloc` allocano nuovo spazio di memoria; i rispettivi prototipi sono:

<pre>#include <stdlib.h> void *calloc(size_t nmemb, size_t size)</pre>	Alloca un'area di memoria inizializzata a 0.
<pre>void *malloc(size_t size)</pre>	Alloca un'area di memoria non inizializzata.

Entrambe le funzioni restituiscono il puntatore alla zona di memoria allocata in caso di successo e NULL in caso di fallimento, nel qual caso <code>errno</code> assumerà il valore <code>ENOMEM</code> .

In genere si usano `malloc` e `calloc` per allocare dinamicamente un'area di memoria.¹⁴ Dato che i puntatori ritornati sono di tipo generico non è necessario effettuare un cast per assegnarli a puntatori al tipo di variabile per la quale si effettua l'allocazione, inoltre le funzioni garantiscono che i puntatori siano allineati correttamente per tutti i tipi di dati; ad esempio sulle macchine a 32 bit in genere sono allineati a multipli di 4 byte e sulle macchine a 64 bit a multipli di 8 byte.

Nel caso di `calloc` l'area di memoria viene allocata nello *heap* come un vettore di `nmemb` membri di `size` byte di dimensione, e preventivamente inizializzata a zero, nel caso di `malloc` invece vengono semplicemente allocati `size` byte e l'area di memoria non viene inizializzata.

¹⁴queste funzioni presentano un comportamento diverso fra la *glibc* e la *uClib* quando il valore di `size` è nullo. Nel primo caso viene comunque restituito un puntatore valido, anche se non è chiaro a cosa esso possa fare riferimento, nel secondo caso viene restituito NULL. Il comportamento è analogo con `realloc(NULL, 0)`.

Una volta che non sia più necessaria la memoria allocata dinamicamente deve essere esplicitamente rilasciata usando la funzione `free`,¹⁵ il suo prototipo è:

```
#include <stdlib.h>
void free(void *ptr)
```

Disalloca un'area di memoria precedentemente allocata.

La funzione non ritorna nulla e non riporta errori.

Questa funzione vuole come argomento `ptr` il puntatore restituito da una precedente chiamata ad una qualunque delle funzioni di allocazione che non sia già stato liberato da un'altra chiamata a `free`. Se il valore di `ptr` è `NULL` la funzione non fa niente, mentre se l'area di memoria era già stata liberata da una precedente chiamata il comportamento della funzione è dichiarato indefinito, ma in genere comporta la corruzione dei dati di gestione dell'allocazione, che può dar luogo a problemi gravi, ad esempio un *segmentation fault* in una successiva chiamata di una di queste funzioni.

Dato che questo errore, chiamato in gergo *double free*, è abbastanza frequente, specie quando si manipolano vettori di puntatori, e dato che le conseguenze possono essere pesanti ed inaspettate, si suggerisce come soluzione precauzionale di assegnare sempre a `NULL` ogni puntatore su cui sia stata eseguita `free` immediatamente dopo l'esecuzione della funzione. In questo modo, dato che con un puntatore nullo `free` non esegue nessuna operazione, si evitano i problemi del *double free*.

Infine la funzione `realloc` consente di modificare, in genere di aumentare, la dimensione di un'area di memoria precedentemente allocata; il suo prototipo è:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size)
```

Cambia la dimensione di un'area di memoria precedentemente allocata.

La funzione ritorna il puntatore alla zona di memoria allocata in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà il valore `ENOMEM`.

La funzione vuole come primo argomento il puntatore restituito da una precedente chiamata a `malloc` o `calloc` e come secondo argomento la nuova dimensione (in byte) che si intende ottenere. Se si passa per `ptr` il valore `NULL` allora la funzione si comporta come `malloc`.¹⁶

La funzione si usa ad esempio quando si deve far crescere la dimensione di un vettore. In questo caso se è disponibile dello spazio adiacente al precedente la funzione lo utilizza, altrimenti rialloca altrove un blocco della dimensione voluta, copiandoci automaticamente il contenuto; lo spazio aggiunto non viene inizializzato. Se la funzione fallisce l'area di memoria originale non viene assolutamente toccata.

Si deve sempre avere ben presente il fatto che il blocco di memoria restituito da `realloc` può non essere un'estensione di quello che gli si è passato in ingresso; per questo si dovrà *sempre* eseguire la riassegnazione di `ptr` al valore di ritorno della funzione, e reinizializzare o provvedere ad un adeguato aggiornamento di tutti gli altri puntatori all'interno del blocco di dati ridimensionato.

La *glibc* ha un'implementazione delle funzioni di allocazione che è controllabile dall'utente attraverso alcune variabili di ambiente (vedi sez. 2.3.3), in particolare diventa possibile tracciare questo tipo di errori usando la variabile di ambiente `MALLOC_CHECK_` che quando viene

¹⁵le *glibc* provvedono anche una funzione `cfree` definita per compatibilità con SunOS, che è deprecata.

¹⁶questo è vero per Linux e l'implementazione secondo lo standard ANSI C, ma non è vero per alcune vecchie implementazioni, inoltre alcune versioni delle librerie del C consentivano di usare `realloc` anche per un puntatore liberato con `free` purché non ci fossero state nel frattempo altre chiamate a funzioni di allocazione, questa funzionalità è totalmente deprecata e non è consentita sotto Linux.

definita mette in uso una versione meno efficiente delle funzioni suddette, che però è più tollerante nei confronti di piccoli errori come quello dei *double free* o i *buffer overrun* di un byte.¹⁷ In particolare:

- se la variabile è posta a 0 gli errori vengono ignorati;
- se la variabile è posta a 1 viene stampato un avviso sullo *standard error* (vedi sez. 5.1.1);
- se la variabile è posta a 2 viene chiamata la funzione **abort** (vedi sez. 7.3.4), che in genere causa l'immediata terminazione del programma;
- se la variabile è posta a 3 viene stampato l'avviso e chiamata **abort**.

L'errore di programmazione più comune e più difficile da risolvere che si incontra con le funzioni di allocazione è quando non viene opportunamente liberata la memoria non più utilizzata, quello che in inglese viene chiamato *memory leak*, cioè una *perdita di memoria*.

Un caso tipico che illustra il problema è quello in cui in una propria funzione si alloca della memoria per uso locale senza liberarla prima di uscire. La memoria resta così allocata fino alla terminazione del processo. Chiamate ripetute alla stessa funzione continueranno ad effettuare altre allocazioni, che si accumuleranno causando a lungo andare un esaurimento della memoria disponibile e la probabile impossibilità di proseguire l'esecuzione del programma.

Il problema è che l'esaurimento della memoria può avvenire in qualunque momento, in corrispondenza ad una qualunque chiamata di **malloc** che può essere in una sezione del codice che non ha alcuna relazione con la funzione che contiene l'errore. Per questo motivo è sempre molto difficile trovare un *memory leak*. In C e C++ il problema è particolarmente sentito. In C++, per mezzo della programmazione ad oggetti, il problema dei *memory leak* si può notevolmente ridimensionare attraverso l'uso accurato di appositi oggetti come gli *smartpointers*. Questo però in genere va a scapito delle prestazioni dell'applicazione in esecuzione.

Per limitare l'impatto di questi problemi, e semplificare la ricerca di eventuali errori, l'implementazione delle funzioni di allocazione nella *glibc* mette a disposizione una serie di funzionalità che permettono di tracciare le allocazioni e le disallocazioni, e definisce anche una serie di possibili *hook* (*ganci*) che permettono di sostituire alle funzioni di libreria una propria versione (che può essere più o meno specializzata per il debugging). Esistono varie librerie che forniscono dei sostituti opportuni delle funzioni di allocazione in grado, senza neanche ricompilare il programma,¹⁸ di eseguire diagnostiche anche molto complesse riguardo l'allocazione della memoria. Vedremo alcune delle funzionalità di ausilio presenti nella *glibc* in sez. 2.2.5.

Una possibile alternativa all'uso di **malloc**, per evitare di soffrire dei problemi di *memory leak* descritti in precedenza, è di allocare la memoria nel segmento di *stack* della funzione corrente invece che nello *heap*. Per farlo si può usare la funzione **alloca**, la cui sintassi è identica a quella di **malloc**; il suo prototipo è:

```
#include <stdlib.h>
void *alloca(size_t size)
```

Alloca un'area di memoria nello *stack*.

La funzione ritorna il puntatore alla zona di memoria allocata, in caso di errore il comportamento è indefinito.

La funzione **alloca** la quantità di memoria (non inizializzata) richiesta dall'argomento **size** nel segmento di *stack* della funzione chiamante. Con questa funzione non è più necessario liberare la memoria allocata, e quindi non esiste un analogo della **free**, in quanto essa viene rilasciata automaticamente al ritorno della funzione.

¹⁷uno degli errori più comuni, causato ad esempio dalla scrittura di una stringa di dimensione pari a quella del buffer, in cui ci si dimentica dello zero di terminazione finale.

¹⁸esempi sono *Dmalloc* <http://dmalloc.com/> di Gray Watson ed *Electric Fence* di Bruce Perens.

Come è evidente questa funzione ha alcuni vantaggi interessanti, anzitutto permette di evitare alla radice i problemi di *memory leak*, dato che non serve più la deallocazione esplicita; inoltre la deallocazione automatica funziona anche quando si usa `longjmp` per uscire da una subroutine con un salto non locale da una funzione (vedi sez. 2.4.3). Un altro vantaggio è che in Linux la funzione è molto più veloce di `malloc` e non viene sprecato spazio, infatti non è necessario gestire un pool di memoria da riservare e si evitano così anche i problemi di frammentazione di quest'ultimo, che comportano inefficienze sia nell'allocazione della memoria che nell'esecuzione dell'allocazione.

Gli svantaggi sono che questa funzione non è disponibile su tutti gli Unix, e non è inserita né nello standard POSIX né in SUSv3 (ma è presente in BSD), il suo utilizzo quindi limita la portabilità dei programmi. Inoltre la funzione non può essere usata nella lista degli argomenti di una funzione, perché lo spazio verrebbe allocato nel mezzo degli stessi. Inoltre non è chiaramente possibile usare `alloca` per allocare memoria che deve poi essere usata anche al di fuori della funzione in cui essa viene chiamata, dato che all'uscita dalla funzione lo spazio allocato diventerebbe libero, e potrebbe essere sovrascritto all'invocazione di nuove funzioni. Questo è lo stesso problema che si può avere con le variabili automatiche, su cui torneremo in sez. 2.4.1.

Infine non esiste un modo di sapere se l'allocazione ha avuto successo, la funzione infatti viene realizzata inserendo del codice *inline* nel programma¹⁹ che si limita a modificare il puntatore nello *stack* e non c'è modo di sapere se se ne sono superate le dimensioni, per cui in caso di fallimento nell'allocazione il comportamento del programma può risultare indefinito, dando luogo ad una *segment violation* la prima volta che si cerchi di accedere alla memoria non effettivamente disponibile.

Le due funzioni seguenti vengono utilizzate soltanto quando è necessario effettuare direttamente la gestione della memoria associata allo spazio dati di un processo,²⁰ per poterle utilizzare è necessario definire una della macro di funzionalità (vedi sez. 1.3.7) fra `_BSD_SOURCE`, `_SVID_SOURCE` e `_XOPEN_SOURCE` (ad un valore maggiore o uguale di 500). La prima funzione è `brk`, ed il suo prototipo è:

```
#include <unistd.h>
int brk(void *addr)
```

Sposta la fine del segmento dati del processo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà il valore `ENOMEM`.

La funzione è un'interfaccia all'omonima *system call* ed imposta l'indirizzo finale del segmento dati di un processo (più precisamente dello *heap*) all'indirizzo specificato da `addr`. Quest'ultimo deve essere un valore ragionevole e la dimensione totale non deve comunque eccedere un eventuale limite (vedi sez. 6.3.2) sulle dimensioni massime del segmento dati del processo.

Il valore di ritorno della funzione fa riferimento alla versione fornita dalla *glibc*, in realtà in Linux la *system call* corrispondente restituisce come valore di ritorno il nuovo valore della fine del segmento dati in caso di successo e quello corrente in caso di fallimento, è la funzione di interfaccia usata dalla *glibc* che fornisce i valori di ritorno appena descritti; se si usano librerie diverse questo potrebbe non accadere.

¹⁹questo comporta anche il fatto che non è possibile sostituirla con una propria versione o modificarne il comportamento collegando il proprio programma con un'altra libreria.

²⁰Le due funzioni sono state definite con BSD 4.3, sono marcate obsolete in SUSv2 e non fanno parte delle librerie standard del C e mentre sono state esplicitamente rimosse dallo standard POSIX.1-2001.

Una seconda funzione per la manipolazione diretta delle dimensioni del segmento dati²¹ è `sbrk`, ed il suo prototipo è:

<pre>#include <unistd.h> void *sbrk(intptr_t increment)</pre>	Incrementa la dimensione del segmento dati del processo.
La funzione ritorna il puntatore all'inizio della nuova zona di memoria allocata in caso di successo e NULL per un errore, nel qual caso <code>errno</code> assumerà il valore <code>ENOMEM</code> .	

La funzione incrementa la dimensione dello *heap* di un programma del valore indicato dall'argomento `increment`, restituendo il nuovo indirizzo finale dello stesso. L'argomento è definito come di tipo `intptr_t`, ma a seconda della versione delle librerie e del sistema può essere indicato con una serie di tipi equivalenti come `ptrdiff_t`, `ssize_t`, `int`. Se invocata con un valore nullo la funzione permette di ottenere l'attuale posizione della fine del segmento dati.

Queste due funzioni sono state deliberatamente escluse dallo standard POSIX.1 dato che per i normali programmi è sempre opportuno usare le funzioni di allocazione standard descritte in precedenza, a meno di non voler realizzare per proprio conto un diverso meccanismo di gestione della memoria del segmento dati.

2.2.4 Il controllo della memoria virtuale

Come spiegato in sez. 2.2.1 il kernel gestisce la memoria virtuale in maniera trasparente ai processi, decidendo quando rimuovere pagine dalla memoria per metterle nell'area di *swap*, sulla base dell'utilizzo corrente da parte dei vari processi.

Nell'uso comune un processo non deve preoccuparsi di tutto ciò, in quanto il meccanismo della paginazione riporta in RAM, ed in maniera trasparente, tutte le pagine che gli occorrono; esistono però esigenze particolari in cui non si vuole che questo meccanismo si attivi. In generale i motivi per cui si possono avere di queste necessità sono due:

- *La velocità.* Il processo della paginazione è trasparente solo se il programma in esecuzione non è sensibile al tempo che occorre a riportare la pagina in memoria; per questo motivo processi critici che hanno esigenze di tempo reale o tolleranze critiche nelle risposte (ad esempio processi che trattano campionamenti sonori) possono non essere in grado di sopportare le variazioni della velocità di accesso dovuta alla paginazione. In certi casi poi un programmatore può conoscere meglio dell'algoritmo di allocazione delle pagine le esigenze specifiche del suo programma e decidere quali pagine di memoria è opportuno che restino in memoria per un aumento delle prestazioni. In genere queste sono esigenze particolari e richiedono anche un aumento delle priorità in esecuzione del processo (vedi sez. 3.3.3).
- *La sicurezza.* Se si hanno password o chiavi segrete in chiaro in memoria queste possono essere portate su disco dal meccanismo della paginazione. Questo rende più lungo il periodo di tempo in cui detti segreti sono presenti in chiaro e più complessa la loro cancellazione: un processo infatti può cancellare la memoria su cui scrive le sue variabili, ma non può toccare lo spazio disco su cui una pagina di memoria può essere stata salvata. Per questo motivo di solito i programmi di crittografia richiedono il blocco di alcune pagine di memoria.

Per ottenere informazioni sulle modalità in cui un programma sta usando la memoria virtuale è disponibile una apposita funzione di sistema, `mincore`, che però non è standardizzata

²¹in questo caso si tratta soltanto di una funzione di libreria, anche se basata sulla stessa *system call*.

da POSIX e pertanto non è disponibile su tutte le versioni di kernel unix-like;²² il suo prototipo è:

```
#include <unistd.h>
#include <sys/mman.h>
int mincore(void *addr, size_t length, unsigned char *vec)
    Ritorna lo stato delle pagine di memoria occupate da un processo.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN il kernel è temporaneamente non in grado di fornire una risposta.
 EFAULT `vec` punta ad un indirizzo non valido.
 EINVAL `addr` non è un multiplo delle dimensioni di una pagina.
 ENOMEM o `addr+length` eccede la dimensione della memoria usata dal processo o l'intervallo di indirizzi specificato non è mappato.

La funzione permette di ottenere le informazioni sullo stato della mappatura della memoria per il processo chiamante, specificando l'intervallo da esaminare con l'indirizzo iniziale, indicato con l'argomento `addr`, e la lunghezza, indicata con l'argomento `length`. L'indirizzo iniziale deve essere un multiplo delle dimensioni di una pagina, mentre la lunghezza può essere qualunque, fintanto che si resta nello spazio di indirizzi del processo,²³ ma il risultato verrà comunque fornito per l'intervallo compreso fino al multiplo successivo.

I risultati della funzione vengono forniti nel vettore puntato da `vec`, che deve essere allocato preventivamente e deve essere di dimensione sufficiente a contenere tanti byte quante sono le pagine contenute nell'intervallo di indirizzi specificato, la dimensione cioè deve essere almeno pari a $(length+PAGE_SIZE-1)/PAGE_SIZE$. Al ritorno della funzione il bit meno significativo di ciascun byte del vettore sarà acceso se la pagina di memoria corrispondente è al momento residente in memoria, o cancellato altrimenti. Il comportamento sugli altri bit è indefinito, essendo questi al momento riservati per usi futuri. Per questo motivo in genere è comunque opportuno inizializzare a zero il contenuto del vettore, così che le pagine attualmente residenti in memoria saranno indicate da un valore non nullo del byte corrispondente.

Dato che lo stato della memoria di un processo può cambiare continuamente, il risultato di `mincore` è assolutamente provvisorio e lo stato delle pagine potrebbe essere già cambiato al ritorno stesso della funzione, a meno che, come vedremo ora, non si sia attivato il meccanismo che forza il mantenimento di una pagina sulla memoria.

Il meccanismo che previene la paginazione di parte della memoria virtuale di un processo è chiamato *memory locking* (o *blocco della memoria*). Il blocco è sempre associato alle pagine della memoria virtuale del processo, e non al segmento reale di RAM su cui essa viene mantenuta. La regola è che se un segmento di RAM fa da supporto ad almeno una pagina bloccata allora esso viene escluso dal meccanismo della paginazione. I blocchi non si accumulano, se si blocca due volte la stessa pagina non è necessario sbloccarla due volte, una pagina o è bloccata oppure no.

Il *memory lock* persiste fintanto che il processo che detiene la memoria bloccata non la sblocca. Chiaramente la terminazione del processo comporta anche la fine dell'uso della sua memoria virtuale, e quindi anche di tutti i suoi *memory lock*. Inoltre i *memory lock* non sono ereditati dai processi figli, ma siccome Linux usa il *copy on write* (vedi sez. 3.1.3) gli indirizzi virtuali del figlio sono mantenuti sullo stesso segmento di RAM del padre, e quindi fintanto che un figlio non scrive su un segmento bloccato, può usufruire del *memory lock* del padre. Infine

²²nel caso di Linux devono essere comunque definite le macro `_BSD_SOURCE` e `_SVID_SOURCE` o `_DEFAULT_SOURCE`.

²³in caso contrario si avrà un errore di `ENOMEM`; fino al kernel 2.6.11 in questo caso veniva invece restituito `EINVAL`, in considerazione che il caso più comune in cui si verifica questo errore è quando si usa per sbaglio un valore negativo di `length`, che nel caso verrebbe interpretato come un intero positivo di grandi dimensioni.

i *memory lock* vengono automaticamente rimossi se si pone in esecuzione un altro programma con `exec` (vedi sez. 3.1.6).

Il sistema pone dei limiti all'ammontare di memoria di un processo che può essere bloccata e al totale di memoria fisica che si può dedicare a questo, lo standard POSIX.1 richiede che sia definita in `unistd.h` la macro `_POSIX_MEMLOCK_RANGE` per indicare la capacità di eseguire il *memory locking*.

Siccome la richiesta di un *memory lock* da parte di un processo riduce la memoria fisica disponibile nel sistema per gli altri processi, questo ha un evidente impatto su tutti gli altri processi, per cui fino al kernel 2.6.9 solo un processo dotato di privilegi amministrativi (la *capability* `CAP_IPC_LOCK`, vedi sez. 9.1.1) aveva la capacità di bloccare una pagina di memoria.

A partire dal kernel 2.6.9 anche un processo normale può bloccare la propria memoria²⁴ ma mentre un processo privilegiato non ha limiti sulla quantità di memoria che può bloccare, un processo normale è soggetto al limite della risorsa `RLIMIT_MEMLOCK` (vedi sez. 6.3.2). In generale poi ogni processo può sbloccare le pagine relative alla propria memoria, se però diversi processi bloccano la stessa pagina questa resterà bloccata fintanto che ci sarà almeno un processo che la blocca.

Le funzioni di sistema per bloccare e sbloccare la paginazione di singole sezioni di memoria sono rispettivamente `mlock` e `munlock`; i loro prototipi sono:

```
#include <sys/mman.h>
int mlock(const void *addr, size_t len)
                                   Blocca la paginazione su un intervallo di memoria.
int munlock(const void *addr, size_t len)
                                   Rimuove il blocco della paginazione su un intervallo di memoria.
```

Entrambe le funzioni ritornano 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN	una parte o tutto l'intervallo richiesto non può essere bloccato per una mancanza temporanea di risorse.
EINVAL	<code>len</code> non è un valore positivo o la somma con <code>addr</code> causa un overflow.
ENOMEM	alcuni indirizzi dell'intervallo specificato non corrispondono allo spazio di indirizzi del processo o con <code>mlock</code> si è superato il limite di <code>RLIMIT_MEMLOCK</code> per un processo non privilegiato (solo per kernel a partire dal 2.6.9) o si è superato il limite di regioni di memoria con attributi diversi.
EPERM	il processo non è privilegiato (per kernel precedenti il 2.6.9) o si ha un limite nullo per <code>RLIMIT_MEMLOCK</code> e il processo non è privilegiato (per kernel a partire dal 2.6.9).

Le due funzioni permettono rispettivamente di bloccare e sbloccare la paginazione per l'intervallo di memoria iniziante all'indirizzo `addr` e lungo `len` byte. Al ritorno di `mlock` tutte le pagine che contengono una parte dell'intervallo bloccato sono garantite essere in RAM e vi verranno mantenute per tutta la durata del blocco. Con kernel diversi da Linux si può ottenere un errore di `EINVAL` se `addr` non è un multiplo della dimensione delle pagine di memoria, pertanto se si ha a cuore la portabilità si deve avere cura di allinearne correttamente il valore. Il blocco viene rimosso chiamando `munlock`.

Altre due funzioni di sistema, `mlockall` e `munlockall`, consentono di bloccare genericamente la paginazione per l'intero spazio di indirizzi di un processo. I prototipi di queste funzioni sono:

²⁴la funzionalità è stata introdotta per non essere costretti a dare privilegi eccessivi a programmi di crittografia, che necessitano di questa funzionalità, ma che devono essere usati da utenti normali.


```
#include <sys/mman.h>
int mlockall(int flags)
    Blocca la paginazione per lo spazio di indirizzi del processo corrente.
int munlockall(void)
    Sblocca la paginazione per lo spazio di indirizzi del processo corrente.
```

Codici di ritorno ed errori sono gli stessi di `mlock` e `munlock`, tranne per `EINVAL` che viene restituito solo se si è specificato con `mlockall` un valore sconosciuto per `flags`.

L'argomento `flags` di `mlockall` permette di controllarne il comportamento; esso deve essere specificato come maschera binaria dei valori espressi dalle costanti riportate in tab. 2.2.

Valore	Significato
MCL_CURRENT	blocca tutte le pagine correntemente mappate nello spazio di indirizzi del processo.
MCL_FUTURE	blocca tutte le pagine che verranno mappate nello spazio di indirizzi del processo.
MCL_ONFAULT	esegue il blocco delle pagine selezionate solo quando vengono utilizzate (dal kernel 4.4).

Tabella 2.2: Valori e significato dell'argomento `flags` della funzione `mlockall`.

Con `mlockall` si possono bloccare tutte le pagine mappate nello spazio di indirizzi del processo, sia che comprendano il segmento di testo, di dati, lo *stack*, lo *heap* e pure le funzioni di libreria chiamate, i file mappati in memoria, i dati del kernel mappati in *user space*, la memoria condivisa. L'uso dell'argomento `flags` permette di selezionare con maggior finezza le pagine da bloccare, ad esempio usando `MCL_FUTURE` ci si può limitare a tutte le pagine allocate a partire dalla chiamata della funzione, mentre `MCL_CURRENT` blocca tutte quelle correntemente mappate. L'uso di `munlockall` invece sblocca sempre tutte le pagine di memoria correntemente mappate nello spazio di indirizzi del programma.

A partire dal kernel 4.4 alla funzione `mlockall` è stato aggiunto un altro flag, `MCL_ONFAULT`, che può essere abbinato a entrambi gli altri due flag, e consente di modificare il comportamento della funzione per ottenere migliori prestazioni.

Il problema che si presenta infatti è che eseguire un *memory lock* per un intervallo ampio di memoria richiede che questa venga comunque allocata in RAM, con altrettanti *page fault* che ne assicurino la presenza; questo vale per tutto l'intervallo e può avere un notevole costo in termini di prestazioni, anche quando poi, nell'esecuzione del programma, venisse usata solo una piccola parte dello stesso. L'uso di `MCL_ONFAULT` previene il *page faulting* immediato di tutto l'intervallo, le pagine dell'intervallo verranno bloccate, ma solo quando un *page fault* dovuto all'accesso ne richiede l'allocazione effettiva in RAM.

Questo stesso comportamento non è ottenibile con `mlock`, che non dispone di un argomento `flag` che consenta di richiederlo, per questo sempre con il kernel 4.4 è stata aggiunta una ulteriore funzione di sistema, `mlock2`, il cui prototipo è:

```
#include <sys/mman.h>
int mlock2(const void *addr, size_t len, int flags)
    Blocca la paginazione su un intervallo di memoria.
```

Le funzione ritornano 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assume gli stessi valori di `mlock` con l'aggiunta di un possibile `EINVAL` anche se si è indicato un valore errato di `flags`.

Indicando un valore nullo per `flags` il comportamento della funzione è identico a quello di `mlock`, l'unico altro valore possibile è `MLOCK_ONFAULT` che ha lo stesso effetto sull'allocazione delle pagine in RAM già descritto per `MCL_ONFAULT`.

Si tenga presente che un processo *real-time* che intende usare il *memory locking* con `mlockall` per prevenire l'avvenire di un eventuale *page fault* ed il conseguente rallentamento (probabilmente inaccettabile) dei tempi di esecuzione, deve comunque avere delle accortezze. In particolare si deve assicurare di aver preventivamente bloccato una quantità di spazio nello *stack* sufficiente a garantire l'esecuzione di tutte le funzioni che hanno i requisiti di criticità sui tempi. Infatti, anche usando `MCL_FUTURE`, in caso di allocazione di una nuova pagina nello *stack* durante l'esecuzione di una funzione (precedentemente non usata e quindi non bloccata) si potrebbe avere un *page fault*.

In genere si ovvia a questa problematica chiamando inizialmente una funzione che definisca una quantità sufficientemente ampia di variabili automatiche (che si ricordi vengono allocate nello *stack*) e ci scriva, in modo da esser sicuri che le corrispondenti pagine vengano mappate nello spazio di indirizzi del processo, per poi bloccarle. La scrittura è necessaria perché il kernel usa il meccanismo di *copy on write* (vedi sez. 3.1.3) e le pagine potrebbero non essere allocate immediatamente.

2.2.5 Gestione avanzata dell'allocazione della memoria

La trattazione delle funzioni di allocazione di sez. 2.2.3 si è limitata a coprire le esigenze generiche di un programma, in cui non si hanno dei requisiti specifici e si lascia il controllo delle modalità di allocazione alle funzioni di libreria. Tuttavia esistono una serie di casi in cui può essere necessario avere un controllo più dettagliato delle modalità con cui la memoria viene allocata; nel qual caso potranno venire in aiuto le funzioni trattate in questa sezione.

Le prime funzioni che tratteremo sono quelle che consentono di richiedere di allocare un blocco di memoria “allineato” ad un multiplo una certa dimensione. Questo tipo di esigenza emerge usualmente quando si devono allocare dei buffer da utilizzare per eseguire dell'I/O diretto su dispositivi a blocchi. In questo caso infatti il trasferimento di dati viene eseguito per blocchi di dimensione fissa, ed è richiesto che l'indirizzo di partenza del buffer sia un multiplo intero di questa dimensione, usualmente 512 byte. In tal caso l'uso di `malloc` non è sufficiente, ed occorre utilizzare una funzione specifica.

Tradizionalmente per rispondere a questa esigenza sono state create due funzioni diverse, `memalign` e `valloc`, oggi obsolete, cui si aggiunge `pvalloc` come estensione GNU, anch'essa obsoleta; i rispettivi prototipi sono:

```
#include <malloc.h>
void *valloc(size_t size)
    Alloca un blocco di memoria allineato alla dimensione di una pagina di memoria.
void *memalign(size_t boundary, size_t size)
    Alloca un blocco di memoria allineato ad un multiplo di boundary.
void *pvalloc(size_t size)
    Alloca un blocco di memoria allineato alla dimensione di una pagina di memoria.
```

Entrambe le funzioni ritornano un puntatore al blocco di memoria allocato in caso di successo e NULL in caso di errore, nel qual caso `errno` assumerà uno dei valori:

```
EINVAL    boundary non è una potenza di due.
ENOMEM    non c'è memoria sufficiente per l'allocazione.
```

Le funzioni restituiscono il puntatore al buffer di memoria allocata di dimensioni pari a `size`, che per `memalign` sarà un multiplo di `boundary` mentre per `valloc` un multiplo della dimensione di una pagina di memoria; lo stesso vale per `pvalloc` che però arrotonda automaticamente la dimensione dell'allocazione al primo multiplo di una pagina. Nel caso della versione fornita dalla *glibc* la memoria allocata con queste funzioni deve essere liberata con `free`, cosa che non è detto accada con altre implementazioni.

Nessuna delle due funzioni ha una chiara standardizzazione e nessuna delle due compare in POSIX.1, inoltre ci sono indicazioni discordi sui file che ne contengono la definizione;²⁵ per questo motivo il loro uso è sconsigliato, essendo state sostituite dalla nuova `posix_memalign`, che è stata standardizzata in POSIX.1d e disponibile dalla *glibc* 2.1.91; il suo prototipo è:

```
#include <stdlib.h>
posix_memalign(void **memptr, size_t alignment, size_t size)
    Alloca un buffer di memoria allineato ad un multiplo di alignment.
```

Entrambe le funzioni ritornano un puntatore al blocco di memoria allocato in caso di successo e NULL in caso di errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL alignment non è potenza di due o un multiplo di `sizeof(void *)`.

ENOMEM non c'è memoria sufficiente per l'allocazione.

La funzione restituisce il puntatore al buffer allocato di dimensioni pari a `size` nella variabile (di tipo `void *`) posta all'indirizzo indicato da `memptr`. La funzione fallisce nelle stesse condizioni delle due funzioni precedenti, ma a loro differenza restituisce direttamente come valore di ritorno il codice di errore. Come per le precedenti la memoria allocata con `posix_memalign` deve essere disallocata con `free`, che in questo caso però è quanto richiesto dallo standard.

Dalla versione 2.16 della *glibc* è stata aggiunta anche la funzione `aligned_alloc`, prevista dallo standard C11 (e disponibile definendo `_ISOC11_SOURCE`), il cui prototipo è:

```
#include <malloc.h>
void *aligned_alloc(size_t alignment, size_t size)
    Alloca un blocco di memoria allineato ad un multiplo di alignment.
```

La funzione ha gli stessi valori di ritorno e codici di errore di `memalign`.

La funzione è identica a `memalign` ma richiede che `size` sia un multiplo di `alignment`. Infine si tenga presente infine che nessuna di queste funzioni inizializza il buffer di memoria allocato, il loro comportamento cioè è analogo, allineamento a parte, a quello di `malloc`.

Un secondo caso in cui risulta estremamente utile poter avere un maggior controllo delle modalità di allocazione della memoria è quello in cui cercano errori di programmazione. Esempi di questi errori sono i *double free*, o i cosiddetti *buffer overrun*, cioè le scritture su un buffer oltre le dimensioni della sua allocazione,²⁶ o i classici *memory leak*.

Abbiamo visto in sez. 2.2.4 come una prima funzionalità di ausilio nella ricerca di questi errori sia l'uso della variabile di ambiente `MALLOC_CHECK_`. Una modalità alternativa per effettuare dei controlli di consistenza sullo stato delle allocazioni di memoria eseguite con `malloc`, anche questa fornita come estensione specifica (e non standard) della *glibc*, è quella di utilizzare la funzione `mcheck`, che deve essere chiamata prima di eseguire qualunque allocazione con `malloc`; il suo prototipo è:

```
#include <mcheck.h>
int mcheck(void (*abortfn) (enum mcheck_status status))
    Attiva i controlli di consistenza delle allocazioni di memoria.
```

La funzione ritorna 0 in caso di successo e -1 per un errore; `errno` non viene impostata.

La funzione consente di registrare una funzione di emergenza che verrà eseguita tutte le volte che, in una successiva esecuzione di `malloc`, venissero trovate delle inconsistenze, come

²⁵secondo SUSv2 `valloc` è definita in `stdlib.h`, mentre sia la *glibc* che le precedenti *libc4* e *libc5* la dichiarano in `malloc.h`, lo stesso vale per `memalign` che in alcuni sistemi è dichiarata in `stdlib.h`.

²⁶entrambe queste operazioni causano in genere la corruzione dei dati di controllo delle funzioni di allocazione, che vengono anch'essi mantenuti nello *heap* per tenere traccia delle zone di memoria allocata.

delle operazioni di scrittura oltre i limiti dei buffer allocati. Per questo motivo la funzione deve essere chiamata prima di qualunque allocazione di memoria, altrimenti fallirà.

Se come primo argomento di `mcheck` si passa `NULL` verrà utilizzata una funzione predefinita che stampa un messaggio di errore ed invoca la funzione `abort` (vedi sez. 7.3.4), altrimenti si dovrà creare una funzione personalizzata in grado di ricevere il tipo di errore ed agire di conseguenza.

Nonostante la scarsa leggibilità del prototipo si tratta semplicemente di definire una funzione di tipo `void abortfn(enum mcheck_status status)`, che non deve restituire nulla e che deve avere un unico argomento di tipo `mcheck_status`. In caso di errore la funzione verrà eseguita ricevendo un opportuno valore di `status` che è un tipo enumerato che può assumere soltanto i valori di tab. 2.3 che indicano la tipologia di errore riscontrata.

Valore	Significato
MCHECK_OK	Riportato a <code>mprobe</code> se nessuna inconsistenza è presente.
MCHECK_DISABLED	Riportato a <code>mprobe</code> se si è chiamata <code>mcheck</code> dopo aver già usato <code>malloc</code> .
MCHECK_HEAD	I dati immediatamente precedenti il buffer sono stati modificati, avviene in genere quando si decrementa eccessivamente il valore di un puntatore scrivendo poi prima dell'inizio del buffer.
MCHECK_TAIL	I dati immediatamente seguenti il buffer sono stati modificati, succede quando si va scrivere oltre la dimensione corretta del buffer.
MCHECK_FREE	Il buffer è già stato disallocato.

Tabella 2.3: Valori dello stato dell'allocazione di memoria ottenibili dalla funzione di terminazione installata con `mcheck`.

Una volta che si sia chiamata `mcheck` con successo si può anche controllare esplicitamente lo stato delle allocazioni senza aspettare un errore nelle relative funzioni utilizzando la funzione `mprobe`, il cui prototipo è:

<pre>#include <mcheck.h> enum mcheck_status mprobe(ptr)</pre>	Esegue un controllo di consistenza delle allocazioni.
La funzione ritorna un codice fra quelli riportati in tab. 2.3 e non ha errori.	

La funzione richiede che si passi come argomento un puntatore ad un blocco di memoria precedentemente allocato con `malloc` o `realloc`, e restituisce lo stesso codice di errore che si avrebbe per la funzione di emergenza ad una successiva chiamata di una funzione di allocazione, e poi i primi due codici che indicano rispettivamente quando tutto è a posto o il controllo non è possibile per non aver chiamato `mcheck` in tempo.

2.3 Argomenti, ambiente ed altre proprietà di un processo

In questa sezione esamineremo le funzioni che permettono di gestire gli argomenti e le opzioni, e quelle che consentono di manipolare ed utilizzare le variabili di ambiente. Accenneremo infine alle modalità con cui si può gestire la localizzazione di un programma modificandone il comportamento a seconda della lingua o del paese a cui si vuole faccia riferimento nelle sue operazioni.

2.3.1 Il formato degli argomenti

Tutti i programmi hanno la possibilità di ricevere argomenti e opzioni quando vengono lanciati e come accennato in sez. 2.1.1 questo viene effettuato attraverso gli argomenti `argc` e `argv` ricevuti nella funzione `main` all'avvio del programma. Questi argomenti vengono passati al programma dalla shell o dal processo che esegue la `exec` (secondo le modalità che vedremo in sez. 3.1.6) quando questo viene messo in esecuzione.

Nel caso più comune il passaggio di argomenti ed opzioni viene effettuato dalla shell, che si incarica di leggere la linea di comando con cui si lancia il programma e di effettuarne la scansione (il cosiddetto *parsing*) per individuare le parole che la compongono, ciascuna delle quali potrà essere considerata un argomento o un'opzione.

Di norma per individuare le parole che andranno a costituire la lista degli argomenti viene usato come carattere di separazione lo spazio o il tabulatore, ma la cosa dipende ovviamente dalle modalità con cui si effettua la scansione e dalle convenzioni adottate dal programma che la esegue: ad esempio la shell consente di proteggere con opportuni caratteri di controllo argomenti che contengono degli spazi evitando di spezzarli in parole diverse.

Figura 2.3: Esempio dei valori di `argv` e `argc` generati nella scansione di una riga di comando.

Indipendentemente da come viene eseguita, il risultato finale della scansione dovrà comunque essere la costruzione del vettore di puntatori `argv` in cui si devono inserire in successione i puntatori alle stringhe costituenti i vari argomenti ed opzioni da passare al programma, e della variabile `argc` che deve essere inizializzata al numero di stringhe contenute in `argv`. Nel caso della shell questo comporta ad esempio che il primo argomento sia sempre il nome del programma. Un esempio di questo meccanismo è mostrato in fig. 2.3, che illustra il risultato della scansione di una riga di comando.

2.3.2 La gestione delle opzioni

In generale un programma Unix riceve da linea di comando sia gli argomenti che le opzioni, queste ultime sono standardizzate per essere riconosciute come tali: un elemento di `argv` successivo al primo che inizia con il carattere “-” e che non sia un singolo “-” o un “--” viene considerato un'opzione. In genere le opzioni sono costituite da una lettera singola (preceduta dal carattere “-”) e possono avere o no un parametro associato.

Un esempio tipico può essere quello mostrato in fig. 2.3. In quel caso le opzioni sono `-r` e `-m` e la prima vuole un parametro mentre la seconda no (`questofile.txt` è un argomento del programma, non un parametro di `-m`).

Per gestire le opzioni all'interno degli argomenti a linea di comando passati in `argv` la libreria standard del C fornisce la funzione `getopt`, che ha il seguente prototipo:

<pre>#include <unistd.h> int getopt(int argc, char * const argv[], const char *optstring)</pre> <p>Esegue la scansione delle opzioni negli argomenti della funzione <code>main</code>.</p>
<p>Ritorna il carattere che segue l'opzione, “.” se manca un parametro all'opzione, “?” se l'opzione è sconosciuta, e <code>-1</code> se non esistono altre opzioni.</p>

Questa funzione prende come argomenti le due variabili `argc` e `argv` che devono essere quelle passate come argomenti di `main` all'esecuzione del programma, ed una stringa `optstring` che indica quali sono le opzioni valide. La funzione effettua la scansione della lista degli argomenti ricercando ogni stringa che comincia con il carattere “-” e ritorna ogni volta che trova un'opzione valida.

La stringa `optstring` indica quali sono le opzioni riconosciute ed è costituita da tutti i caratteri usati per identificare le singole opzioni, se l'opzione ha un parametro al carattere deve essere fatto seguire il carattere di due punti (":"); nel caso di fig. 2.3 ad esempio la stringa di opzioni avrebbe dovuto contenere "r:m".

La modalità di uso di `getopt` è pertanto quella di chiamare più volte la funzione all'interno di un ciclo, fintanto che essa non ritorna il valore `-1` che indica che non ci sono più opzioni. Nel caso si incontri un'opzione non dichiarata in `optstring` viene ritornato il carattere "?" mentre se un'opzione che lo richiede non è seguita da un parametro viene ritornato il carattere ":", infine se viene incontrato il valore "--" la scansione viene considerata conclusa, anche se vi sono altri elementi di `argv` che cominciano con il carattere "-".

Quando `getopt` trova un'opzione fra quelle indicate in `optstring` essa ritorna il valore numerico del carattere, in questo modo si possono eseguire azioni specifiche usando uno `switch`; la funzione inoltre inizializza alcune variabili globali:

- `char *optarg` contiene il puntatore alla stringa parametro dell'opzione.
- `int optind` alla fine della scansione restituisce l'indice del primo elemento di `argv` che non è un'opzione.
- `int opterr` previene, se posto a zero, la stampa di un messaggio di errore in caso di riconoscimento di opzioni non definite.
- `int optopt` contiene il carattere dell'opzione non riconosciuta.

```

1  opterr = 0; /* don't want writing to stderr */
2  while ( (i = getopt(argc, argv, "hp:c:e:")) != -1) {
3      switch (i) {
4          /*
5           * Handling options
6           */
7          case 'h': /* help option */
8              printf("Wrong_-h_option_use\n");
9              usage();
10             return -1;
11             break;
12          case 'c': /* take wait time for children */
13              wait_child = strtol(optarg, NULL, 10); /* convert input */
14              break;
15          case 'p': /* take wait time for children */
16              wait_parent = strtol(optarg, NULL, 10); /* convert input */
17              break;
18          case 'e': /* take wait before parent exit */
19              wait_end = strtol(optarg, NULL, 10); /* convert input */
20              break;
21          case '?': /* unrecognized options */
22              printf("Unrecognized_options_-?c\n", optopt);
23              usage();
24          default: /* should not reached */
25              usage();
26      }
27  }
28  debug("Optind_%d,_argc_%d\n", optind, argc);

```

Figura 2.4: Esempio di codice per la gestione delle opzioni.

In fig. 2.4 si è mostrata la sezione del programma `fork_test.c`, che useremo nel prossimo capitolo per effettuare dei test sulla creazione dei processi, deputata alla decodifica delle opzioni a riga di comando da esso supportate.

Si può notare che si è anzitutto (1) disabilitata la stampa di messaggi di errore per opzioni non riconosciute, per poi passare al ciclo per la verifica delle opzioni (2-27); per ciascuna delle opzioni possibili si è poi provveduto ad un'azione opportuna, ad esempio per le tre opzioni che prevedono un parametro si è effettuata la decodifica del medesimo, il cui indirizzo è contenuto nella variabile `optarg`, avvalorando la relativa variabile (12-14, 15-17 e 18-20). Completato il ciclo troveremo in `optind` l'indice in `argv[]` del primo degli argomenti rimanenti nella linea di comando.

Normalmente `getopt` compie una permutazione degli elementi di `argv` cosicché alla fine della scansione gli elementi che non sono opzioni sono spostati in coda al vettore. Oltre a questa esistono altre due modalità di gestire gli elementi di `argv`; se `optstring` inizia con il carattere “+” (o è impostata la variabile di ambiente `POSIXLY_CORRECT`) la scansione viene fermata non appena si incontra un elemento che non è un'opzione.

L'ultima modalità, usata quando un programma può gestire la mescolanza fra opzioni e argomenti, ma se li aspetta in un ordine definito, si attiva quando `optstring` inizia con il carattere “-”. In questo caso ogni elemento che non è un'opzione viene considerato comunque un'opzione e associato ad un valore di ritorno pari ad 1, questo permette di identificare gli elementi che non sono opzioni, ma non effettua il riordinamento del vettore `argv`.

2.3.3 Le variabili di ambiente

Oltre agli argomenti passati a linea di comando esiste un'altra modalità che permette di trasferire ad un processo delle informazioni in modo da modificarne il comportamento. Ogni processo infatti riceve dal sistema, oltre alle variabili `argv` e `argc` anche un *ambiente* (in inglese *environment*); questo viene espresso nella forma di una lista (chiamata *environment list*) delle cosiddette *variabili di ambiente*, i valori di queste variabili possono essere poi usati dal programma.

Anche in questo caso la lista delle *variabili di ambiente* deve essere costruita ed utilizzata nella chiamata alla funzione `exec` (torneremo su questo in sez. 3.1.6) quando questo viene lanciato. Come per la lista degli argomenti anche questa lista è un vettore di puntatori a caratteri, ciascuno dei quali punta ad una stringa, terminata da un `NULL`. A differenza di `argv[]` in questo caso non si ha una lunghezza del vettore data da un equivalente di `argc`, ma la lista è terminata da un puntatore nullo.

L'indirizzo della lista delle variabili di ambiente è passato attraverso la variabile globale `environ`, che viene definita automaticamente per ciascun processo, e a cui si può accedere attraverso una semplice dichiarazione del tipo:

```
extern char ** environ;
```

un esempio della struttura di questa lista, contenente alcune delle variabili più comuni che normalmente sono definite dal sistema, è riportato in fig. 2.5.

Figura 2.5: Esempio di lista delle variabili di ambiente.

Per convenzione le stringhe che definiscono l'ambiente sono tutte del tipo `NOME=valore` ed in questa forma che le funzioni di gestione che vedremo a breve se le aspettano, se pertanto si dovesse costruire manualmente un ambiente si abbia cura di rispettare questa convenzione. Inoltre alcune variabili, come quelle elencate in fig. 2.5, sono definite dal sistema per essere

usate da diversi programmi e funzioni: per queste c'è l'ulteriore convenzione di usare nomi espressi in caratteri maiuscoli.²⁷

Il kernel non usa mai queste variabili, il loro uso e la loro interpretazione è riservata alle applicazioni e ad alcune funzioni di libreria; in genere esse costituiscono un modo comodo per definire un comportamento specifico senza dover ricorrere all'uso di opzioni a linea di comando o di file di configurazione. È di norma cura della shell, quando esegue un comando, passare queste variabili al programma messo in esecuzione attraverso un uso opportuno delle relative chiamate (si veda sez. 3.1.6).

La shell ad esempio ne usa molte per il suo funzionamento, come `PATH` per indicare la lista delle directory in cui effettuare la ricerca dei comandi o `PS1` per impostare il proprio *prompt*. Alcune di esse, come `HOME`, `USER`, ecc. sono invece definite al login (per i dettagli si veda sez. 8.1.4), ed in genere è cura della propria distribuzione definire le opportune variabili di ambiente in uno script di avvio. Alcune servono poi come riferimento generico per molti programmi, come `EDITOR` che indica l'editor preferito da invocare in caso di necessità. Una in particolare, `LANG`, serve a controllare la localizzazione del programma per adattarlo alla lingua ed alle convenzioni dei vari paesi.

Gli standard POSIX e XPG3 definiscono alcune di queste variabili (le più comuni), come riportato in tab. 2.4. GNU/Linux le supporta tutte e ne definisce anche altre, in particolare poi alcune funzioni di libreria prevedono la presenza di specifiche variabili di ambiente che ne modificano il comportamento, come quelle usate per indicare una localizzazione e quelle per indicare un fuso orario; una lista più completa che comprende queste ed ulteriori variabili si può ottenere con il comando `man 7 environ`.

Variabile	POSIX	XPG3	Linux	Descrizione
<code>USER</code>	•	•	•	Nome utente.
<code>LOGNAME</code>	•	•	•	Nome di login.
<code>HOME</code>	•	•	•	Directory base dell'utente.
<code>LANG</code>	•	•	•	Localizzazione.
<code>PATH</code>	•	•	•	Elenco delle directory dei programmi.
<code>PWD</code>	•	•	•	Directory corrente.
<code>SHELL</code>	•	•	•	Shell in uso.
<code>TERM</code>	•	•	•	Tipo di terminale.
<code>PAGER</code>	•	•	•	Programma per vedere i testi.
<code>EDITOR</code>	•	•	•	Editor preferito.
<code>BROWSER</code>	•	•	•	Browser preferito.
<code>TMPDIR</code>	•	•	•	Directory dei file temporanei.

Tabella 2.4: Esempi delle variabili di ambiente più comuni definite da vari standard.

Lo standard ANSI C prevede l'esistenza di un ambiente, e pur non entrando nelle specifiche di come sono strutturati i contenuti, definisce la funzione `getenv` che permette di ottenere i valori delle variabili di ambiente; il suo prototipo è:

<pre>#include <stdlib.h> char *getenv(const char *name)</pre>	Cerca una variabile di ambiente del processo.
La funzione ritorna il puntatore alla stringa contenente il valore della variabile di ambiente in caso di successo e <code>NULL</code> per un errore.	

La funzione effettua una ricerca nell'ambiente del processo cercando una variabile il cui nome corrisponda a quanto indicato con l'argomento `name`, ed in caso di successo ritorna il puntatore alla stringa che ne contiene il valore, nella forma "`NOME=valore`".

²⁷ma si tratta solo di una convenzione, niente vieta di usare caratteri minuscoli, come avviene in vari casi.

Funzione	ANSI C	POSIX.1	XPG3	SVr4	BSD	Linux
getenv	•	•	•	•	•	•
setenv	—	—	—	—	•	•
unsetenv	—	—	—	—	•	•
putenv	—	opz.	•	—	•	•
clearenv	—	opz.	—	—	—	•

Tabella 2.5: Funzioni per la gestione delle variabili di ambiente.

Oltre a questa funzione di lettura, che è l'unica definita dallo standard ANSI C, nell'evoluzione dei sistemi Unix ne sono state proposte altre, da utilizzare per impostare, modificare e cancellare le variabili di ambiente. Uno schema delle funzioni previste nei vari standard e disponibili in Linux è riportato in tab. 2.5. Tutte le funzioni sono state comunque inserite nello standard POSIX.1-2001, ad eccezione di `clearenv` che è stata rigettata.

In Linux sono definite tutte le funzioni elencate in tab. 2.5,²⁸ anche se parte delle funzionalità sono ridondanti. La prima funzione di manipolazione che prenderemo in considerazione è `putenv`, che consente di aggiungere, modificare e cancellare una variabile di ambiente; il suo prototipo è:

```
int putenv(char *string)
```

Inserisce, modifica o rimuove una variabile d'ambiente.

La funzione ritorna 0 in caso di successo e `-1` per un errore, che può essere solo `ENOMEM`.

La funzione prende come argomento una stringa analoga a quella restituita da `getenv` e sempre nella forma “`NOME=valore`”. Se la variabile specificata (nel caso `NOME`) non esiste la stringa sarà aggiunta all'ambiente, se invece esiste il suo valore sarà impostato a quello specificato dal contenuto di `string` (nel caso `valore`). Se invece si passa come argomento solo il nome di una variabile di ambiente (cioè `string` è nella forma “`NOME`” e non contiene il carattere “`=`”) allora questa, se presente nell'ambiente, verrà cancellata.

Si tenga presente che, seguendo lo standard SUSv2, le *glibc* successive alla versione 2.1.2 aggiungono direttamente `string` nella lista delle variabili di ambiente illustrata in fig. 2.5 sostituendo il relativo puntatore;²⁹ pertanto ogni cambiamento alla stringa in questione si riflette automaticamente sull'ambiente, e quindi si deve evitare di passare a questa funzione una variabile automatica (per evitare i problemi esposti in sez. 2.4.1). Benché non sia richiesto dallo standard, nelle versioni della *glibc* a partire dalla 2.1 la funzione è rientrante (vedi sez. 3.4.3).

Infine quando una chiamata a `putenv` comporta la necessità di creare una nuova versione del vettore `environ` questo sarà allocato automaticamente, ma la versione corrente sarà deallocata solo se anch'essa è risultante da un'allocatione fatta in precedenza da un'altra `putenv`. Questo avviene perché il vettore delle variabili di ambiente iniziale, creato dalla chiamata ad `exec` (vedi sez. 3.1.6) è piazzato nella memoria al di sopra dello *stack*, (vedi fig. 2.2) e non nello *heap* e quindi non può essere deallocato. Inoltre la memoria associata alle variabili di ambiente eliminate non viene liberata.

Come alternativa a `putenv` si può usare la funzione `setenv` che però consente solo di aggiungere o modificare una variabile di ambiente; il suo prototipo è:

²⁸in realtà nelle *libc4* e *libc5* sono definite solo le prime quattro, `clearenv` è stata introdotta con la *glibc* 2.0.

²⁹il comportamento è lo stesso delle vecchie *libc4* e *libc5*; nella *glibc*, dalla versione 2.0 alla 2.1.1, veniva invece fatta una copia, seguendo il comportamento di BSD4.4; dato che questo può dar luogo a perdite di memoria e non rispetta lo standard il comportamento è stato modificato a partire dalla 2.1.2, eliminando anche, sempre in conformità a SUSv2, l'attributo `const` dal prototipo.

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite)
```

Inserisce o modifica una variabile di ambiente.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `name` è NULL o una stringa di lunghezza nulla o che contiene il carattere “=”.
`ENOMEM` non c'è memoria sufficiente per aggiungere una nuova variabile all'ambiente.

La funzione consente di specificare separatamente nome e valore della variabile di ambiente da aggiungere negli argomenti `name` e `value`. Se la variabile è già presente nell'ambiente l'argomento `overwrite` specifica il comportamento della funzione, se diverso da zero sarà sovrascritta, se uguale a zero sarà lasciata immutata. A differenza di `putenv` la funzione esegue delle copie del contenuto degli argomenti `name` e `value` e non è necessario preoccuparsi di allocarli in maniera permanente.

La cancellazione di una variabile di ambiente viene invece gestita esplicitamente con `unsetenv`, il cui prototipo è:

```
#include <stdlib.h>
int unsetenv(const char *name)
```

Rimuove una variabile di ambiente.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `name` è NULL o una stringa di lunghezza nulla o che contiene il carattere “=”.

La funzione richiede soltanto il nome della variabile di ambiente nell'argomento `name`, se la variabile non esiste la funzione ritorna comunque con un valore di successo.³⁰

L'ultima funzione per la gestione dell'ambiente è `clearenv`,³¹ che viene usata per cancellare completamente tutto l'ambiente; il suo prototipo è:

```
#include <stdlib.h>
int clearenv(void)
```

Cancella tutto l'ambiente.

La funzione ritorna 0 in caso di successo e un valore diverso da zero per un errore.

In genere si usa questa funzione in maniera precauzionale per evitare i problemi di sicurezza connessi nel trasmettere ai programmi che si invocano un ambiente che può contenere dei dati non controllati, le cui variabili possono causare effetti indesiderati. Con l'uso della funzione si provvede alla cancellazione di tutto l'ambiente originale in modo da poterne costruirne una versione “sicura” da zero.

2.4 Problematiche di programmazione generica

Benché questo non sia un libro sul linguaggio C, è opportuno affrontare alcune delle problematiche generali che possono emergere nella programmazione con questo linguaggio e di quali precauzioni o accorgimenti occorre prendere per risolverle. Queste problematiche non sono specifiche di sistemi unix-like o *multitasking*, ma avendo trattato in questo capitolo il comportamento dei processi visti come entità a sé stanti, le riportiamo qui.

³⁰questo con le versioni della *glibc* successive la 2.2.2, per le precedenti `unsetenv` era definita come `void` e non restituiva nessuna informazione.

³¹che come accennato è l'unica non presente nello standard POSIX.1-2000, ed è disponibili solo per versioni della *glibc* a partire dalla 2.0; per poterla utilizzare occorre aver definito le macro `_SVID_SOURCE` e `_XOPEN_SOURCE`.

2.4.1 Il passaggio di variabili e valori di ritorno nelle funzioni

Una delle caratteristiche standard del C è che le variabili vengono passate alle funzioni che si invocano in un programma attraverso un meccanismo che viene chiamato *by value*, diverso ad esempio da quanto avviene con il Fortran, dove le variabili sono passate, come suol dirsi, *by reference*, o dal C++ dove la modalità del passaggio può essere controllata con l'operatore `&`.

Il passaggio di una variabile *by value* significa che in realtà quello che viene passato alla funzione è una copia del valore attuale di quella variabile, copia che la funzione potrà modificare a piacere, senza che il valore originale nella funzione chiamante venga toccato. In questo modo non occorre preoccuparsi di eventuali effetti delle operazioni svolte nella funzione stessa sulla variabile passata come argomento.

Questo però va inteso nella maniera corretta. Il passaggio *by value* vale per qualunque variabile, puntatori compresi; quando però in una funzione si usano dei puntatori (ad esempio per scrivere in un buffer) in realtà si va a modificare la zona di memoria a cui essi puntano, per cui anche se i puntatori sono copie, i dati a cui essi puntano saranno sempre gli stessi, e le eventuali modifiche avranno effetto e saranno visibili anche nella funzione chiamante.

Nella maggior parte delle funzioni di libreria e delle *system call* i puntatori vengono usati per scambiare dati (attraverso i buffer o le strutture a cui fanno riferimento) e le variabili normali vengono usate per specificare argomenti; in genere le informazioni a riguardo dei risultati vengono passate alla funzione chiamante attraverso il valore di ritorno. È buona norma seguire questa pratica anche nella programmazione normale.

Talvolta però è necessario che la funzione possa restituire indietro alla funzione chiamante un valore relativo ad uno dei suoi argomenti usato anche in ingresso. Per far questo si usa il cosiddetto *value result argument*, si passa cioè, invece di una normale variabile, un puntatore alla stessa. Gli esempi di questa modalità di passaggio sono moltissimi, ad esempio essa viene usata nelle funzioni che gestiscono i socket (in sez. 15.2), in cui, per permettere al kernel di restituire informazioni sulle dimensioni delle strutture degli indirizzi utilizzate, viene usato proprio questo meccanismo.

Occorre tenere ben presente questa differenza, perché le variabili passate in maniera ordinaria, che vengono inserite nello *stack*, cessano di esistere al ritorno di una funzione, ed ogni loro eventuale modifica all'interno della stessa sparisce con la conclusione della stessa, per poter passare delle informazioni occorre quindi usare un puntatore che faccia riferimento ad un indirizzo accessibile alla funzione chiamante.

Questo requisito di accessibilità è fondamentale, infatti dei possibili problemi che si possono avere con il passaggio dei dati è quello di restituire alla funzione chiamante dei dati che sono contenuti in una variabile automatica. Ovviamente quando la funzione ritorna la sezione dello *stack* che conteneva la variabile automatica (si ricordi quanto detto in sez. 2.2.3) verrà liberata automaticamente e potrà essere riutilizzata all'invocazione di un'altra funzione, con le immaginabili conseguenze, quasi invariabilmente catastrofiche, di sovrapposizione e sovrascrittura dei dati.

Per questo una delle regole fondamentali della programmazione in C è che all'uscita di una funzione non deve restare nessun riferimento alle sue variabili locali. Qualora sia necessario utilizzare delle variabili che devono essere viste anche dalla funzione chiamante queste devono essere allocate esplicitamente, o in maniera statica usando variabili globali o dichiarate come `extern`,³² o dinamicamente con una delle funzioni della famiglia `malloc`, passando opportunamente il relativo puntatore fra le funzioni.

³²la direttiva `extern` informa il compilatore che la variabile che si è dichiarata in una funzione non è da considerarsi locale, ma globale, e per questo allocata staticamente e visibile da tutte le funzioni dello stesso programma.

2.4.2 Il passaggio di un numero variabile di argomenti

Come vedremo nei capitoli successivi, non sempre è possibile specificare un numero fisso di argomenti per una funzione. Lo standard ISO C prevede nella sua sintassi la possibilità di definire delle *variadic function* che abbiano un numero variabile di argomenti, attraverso l'uso nella dichiarazione della funzione dello speciale costrutto "...", che viene chiamato *ellipsis*.

Lo standard però non provvede a livello di linguaggio alcun meccanismo con cui dette funzioni possano accedere ai loro argomenti. L'accesso viene pertanto realizzato a livello della libreria standard del C che fornisce gli strumenti adeguati. L'uso di una *variadic function* prevede quindi tre punti:

- *dichiarare* la funzione come *variadic* usando un prototipo che contenga una *ellipsis*;
- *definire* la funzione come *variadic* usando la stessa *ellipsis*, ed utilizzare le apposite macro che consentono la gestione di un numero variabile di argomenti;
- *invocare* la funzione specificando prima gli argomenti fissi, ed a seguire quelli addizionali.

Lo standard ISO C prevede che una *variadic function* abbia sempre almeno un argomento fisso. Prima di effettuare la dichiarazione deve essere incluso l'apposito *header file* `stdarg.h`; un esempio di dichiarazione è il prototipo della funzione `execl` che vedremo in sez. 3.1.6:

```
int execl(const char *path, const char *arg, ...);
```

in questo caso la funzione prende due argomenti fissi ed un numero variabile di altri argomenti, che andranno a costituire gli elementi successivi al primo del vettore `argv` passato al nuovo processo. Lo standard ISO C richiede inoltre che l'ultimo degli argomenti fissi sia di tipo *self-promoting*³³ il che esclude vettori, puntatori a funzioni e interi di tipo `char` o `short` (con segno o meno). Una restrizione ulteriore di alcuni compilatori è di non dichiarare l'ultimo argomento fisso come variabile di tipo *register*.³⁴

Una volta dichiarata la funzione il secondo passo è accedere ai vari argomenti quando la si va a definire. Gli argomenti fissi infatti hanno un loro nome, ma quelli variabili vengono indicati in maniera generica dalla *ellipsis*. L'unica modalità in cui essi possono essere recuperati è pertanto quella sequenziale, in cui vengono estratti dallo *stack* secondo l'ordine in cui sono stati scritti nel prototipo della funzione.

Per fare questo in `stdarg.h` sono definite delle macro specifiche, previste dallo standard ISO C89, che consentono di eseguire questa operazione. La prima di queste macro è `va_start`, che inizializza opportunamente una lista degli argomenti, la sua definizione è:

```
#include <stdarg.h>
void va_start(va_list ap, last)
```

Inizializza una lista degli argomenti di una funzione *variadic*.

La macro inizializza il puntatore alla lista di argomenti `ap` che deve essere una apposita variabile di tipo `va_list`; il parametro `last` deve indicare il nome dell'ultimo degli argomenti fissi dichiarati nel prototipo della funzione *variadic*.

³³il linguaggio C prevede che quando si mescolano vari tipi di dati, alcuni di essi possano essere *promossi* per compatibilità; ad esempio i tipi `float` vengono convertiti automaticamente a `double` ed i `char` e gli `short` ad `int`. Un tipo *self-promoting* è un tipo che verrebbe promosso a sé stesso.

³⁴la direttiva `register` del compilatore chiede che la variabile dichiarata tale sia mantenuta, nei limiti del possibile, all'interno di un registro del processore; questa direttiva è originaria dell'epoca dai primi compilatori, quando stava al programmatore scrivere codice ottimizzato, riservando esplicitamente alle variabili più usate l'uso dei registri del processore, oggi questa direttiva è in disuso pressoché completo dato che tutti i compilatori sono normalmente in grado di valutare con maggior efficacia degli stessi programmatori quando sia il caso di eseguire questa ottimizzazione.

La seconda macro di gestione delle liste di argomenti di una funzione *variadic* è `va_arg`, che restituisce in successione un argomento della lista; la sua definizione è:

```
#include <stdarg.h>
type va_arg(va_list ap, type)
```

Restituisce il valore del successivo argomento opzionale.

La macro restituisce il valore di un argomento, modificando opportunamente la lista `ap` perché una chiamata successiva restituisca l'argomento seguente. La macro richiede che si specifichi il tipo dell'argomento che si andrà ad estrarre attraverso il parametro `type` che sarà anche il tipo del valore da essa restituito. Si ricordi che il tipo deve essere *self-promoting*.

In generale è perfettamente legittimo richiedere meno argomenti di quelli che potrebbero essere stati effettivamente forniti, per cui nella esecuzione delle `va_arg` ci si può fermare in qualunque momento ed i restanti argomenti saranno ignorati. Se invece si richiedono più argomenti di quelli effettivamente forniti si otterranno dei valori indefiniti. Si avranno risultati indefiniti anche quando si chiama `va_arg` specificando un tipo che non corrisponde a quello usato per il corrispondente argomento.

Infine una volta completata l'estrazione occorre indicare che si sono concluse le operazioni con la macro `va_end`, la cui definizione è:

```
#include <stdarg.h>
void va_end(va_list ap)
```

Conclude l'estrazione degli argomenti di una funzione *variadic*.

Dopo l'uso di `va_end` la variabile `ap` diventa indefinita e successive chiamate a `va_arg` non funzioneranno. Nel caso del gcc l'uso di `va_end` può risultare inutile, ma è comunque necessario usarla per chiarezza del codice, per compatibilità con diverse implementazioni e per eventuali modifiche future a questo comportamento.

Riassumendo la procedura da seguire per effettuare l'estrazione degli argomenti di una funzione *variadic* è la seguente:

1. inizializzare una lista degli argomenti attraverso la macro `va_start`;
2. accedere agli argomenti con chiamate successive alla macro `va_arg`: la prima chiamata restituirà il primo argomento, la seconda il secondo e così via;
3. dichiarare la conclusione dell'estrazione degli argomenti invocando la macro `va_end`.

Si tenga presente che si possono usare anche più liste degli argomenti, ciascuna di esse andrà inizializzata con `va_start` e letta con `va_arg`, e ciascuna potrà essere usata per scandire la lista degli argomenti in modo indipendente. Infine ciascuna scansione dovrà essere terminata con `va_end`.

Un limite di queste macro è che i passi 1) e 3) devono essere eseguiti nel corpo principale della funzione, il passo 2) invece può essere eseguito anche in un'altra funzione, passandole lista degli argomenti `ap`. In questo caso però al ritorno della funzione `va_arg` non può più essere usata (anche se non si era completata l'estrazione) dato che il valore di `ap` risulterebbe indefinito.

Esistono dei casi in cui è necessario eseguire più volte la scansione degli argomenti e poter memorizzare una posizione durante la stessa. In questo caso sembrerebbe naturale copiarsi la lista degli argomenti `ap` con una semplice assegnazione ad un'altra variabile dello stesso tipo. Dato che una delle realizzazioni più comuni di `va_list` è quella di un puntatore nello *stack* all'indirizzo dove sono stati salvati gli argomenti, è assolutamente normale pensare di poter effettuare questa operazione.

In generale però possono esistere anche realizzazioni diverse, ed è per questo motivo che invece che un semplice puntatore, `va_list` è quello che viene chiamato un *tipo opaco*. Si chiamano così quei tipi di dati, in genere usati da una libreria, la cui struttura interna non deve

essere vista dal programma chiamante (da cui deriva il nome opaco) che li devono utilizzare solo attraverso dalle opportune funzioni di gestione.

Per questo motivo una variabile di tipo `va_list` non può essere assegnata direttamente ad un'altra variabile dello stesso tipo, ma lo standard ISO C99³⁵ ha previsto una macro ulteriore che permette di eseguire la copia di una lista degli argomenti:

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src)
    Copia la lista degli argomenti di una funzione variadic.
```

La macro copia l'attuale della lista degli argomenti `src` su una nuova lista `dest`. Anche in questo caso è buona norma chiudere ogni esecuzione di una `va_copy` con una corrispondente `va_end` sul nuovo puntatore alla lista degli argomenti.

La chiamata di una funzione con un numero variabile di argomenti, posto che la si sia dichiarata e definita come tale, non prevede nulla di particolare; l'invocazione è identica alle altre, con gli argomenti, sia quelli fissi che quelli opzionali, separati da virgole. Quello che però è necessario tenere presente è come verranno convertiti gli argomenti variabili.

In Linux gli argomenti dello stesso tipo sono passati allo stesso modo, sia che siano fissi sia che siano opzionali (alcuni sistemi trattano diversamente gli opzionali), ma dato che il prototipo non può specificare il tipo degli argomenti opzionali, questi verranno sempre promossi, pertanto nella ricezione dei medesimi occorrerà tenerne conto (ad esempio un `char` verrà visto da `va_arg` come `int`).

Un altro dei problemi che si devono affrontare con le funzioni con un numero variabile di argomenti è che non esiste un modo generico che permetta di stabilire quanti sono gli argomenti effettivamente passati in una chiamata.

Esistono varie modalità per affrontare questo problema; una delle più immediate è quella di specificare il numero degli argomenti opzionali come uno degli argomenti fissi. Una variazione di questo metodo è l'uso di un argomento fisso per specificare anche il tipo degli argomenti variabili, come fa la stringa di formato per `printf` (vedi sez. 5.3.6).

Infine una ulteriore modalità diversa, che può essere applicata solo quando il tipo degli argomenti lo rende possibile, è quella che prevede di usare un valore speciale per l'ultimo argomento, come fa ad esempio `exec1` che usa un puntatore `NULL` per indicare la fine della lista degli argomenti (vedi sez. 3.1.6).

2.4.3 Il controllo di flusso non locale

Il controllo del flusso di un programma in genere viene effettuato con le varie istruzioni del linguaggio C; fra queste la più bistrattata è il `goto`, che viene deprecato in favore dei costrutti della programmazione strutturata, che rendono il codice più leggibile e mantenibile. Esiste però un caso in cui l'uso di questa istruzione porta all'implementazione più efficiente e più chiara anche dal punto di vista della struttura del programma: quello dell'uscita in caso di errore.

Il C però non consente di effettuare un salto ad una etichetta definita in un'altra funzione, per cui se l'errore avviene in una funzione, e la sua gestione ordinaria è in un'altra, occorre usare quello che viene chiamato un *salto non-locale*. Il caso classico in cui si ha questa necessità, citato sia in [?] che in [?], è quello di un programma nel cui corpo principale vengono letti dei dati in ingresso sui quali viene eseguita, tramite una serie di funzioni di analisi, una scansione dei contenuti, da cui si ottengono le indicazioni per l'esecuzione di opportune operazioni.

³⁵alcuni sistemi che non hanno questa macro provvedono al suo posto `__va_copy` che era il nome proposto in una bozza dello standard.

Dato che l'analisi può risultare molto complessa, ed opportunamente suddivisa in fasi diverse, la rilevazione di un errore nei dati in ingresso può accadere all'interno di funzioni profondamente annidate l'una nell'altra. In questo caso si dovrebbe gestire, per ciascuna fase, tutta la casistica del passaggio all'indietro di tutti gli errori rilevabili dalle funzioni usate nelle fasi successive. Questo comporterebbe una notevole complessità, mentre sarebbe molto più comodo poter tornare direttamente al ciclo di lettura principale, scartando l'input come errato.³⁶

Tutto ciò può essere realizzato proprio con un salto non-locale; questo di norma viene realizzato salvando il contesto dello *stack* nel punto in cui si vuole tornare in caso di errore, e ripristinandolo, in modo da tornare quando serve nella funzione da cui si era partiti. La funzione che permette di salvare il contesto dello *stack* è `setjmp`, il cui prototipo è:

```
#include <setjmp.h>
int setjmp(jmp_buf env)
```

Salva il contesto dello *stack*.

La funzione ritorna 0 quando è chiamata direttamente ed un valore diverso da zero quando ritorna da una chiamata di `longjmp` che usa il contesto salvato in precedenza.

Quando si esegue la funzione il contesto corrente dello *stack* viene salvato nell'argomento `env`, una variabile di tipo `jmp_buf`³⁷ che deve essere stata definita in precedenza. In genere le variabili di tipo `jmp_buf` vengono definite come variabili globali in modo da poter essere viste in tutte le funzioni del programma.

Quando viene eseguita direttamente la funzione ritorna sempre zero, un valore diverso da zero viene restituito solo quando il ritorno è dovuto ad una chiamata di `longjmp` in un'altra parte del programma che ripristina lo *stack* effettuando il salto non-locale. Si tenga conto che il contesto salvato in `env` viene invalidato se la funzione che ha chiamato `setjmp` ritorna, nel qual caso un successivo uso di `longjmp` può comportare conseguenze imprevedibili (e di norma fatali) per il processo.

Come accennato per effettuare un salto non-locale ad un punto precedentemente stabilito con `setjmp` si usa la funzione `longjmp`; il suo prototipo è:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val)
```

Ripristina il contesto dello *stack*.

La funzione non ritorna.

La funzione ripristina il contesto dello *stack* salvato da una chiamata a `setjmp` nell'argomento `env`. Dopo l'esecuzione della funzione il programma prosegue nel codice successivo alla chiamata della `setjmp` con cui si era salvato `env`, che restituirà il valore dell'argomento `val` invece di zero. Il valore dell'argomento `val` deve essere sempre diverso da zero, se si è specificato 0 sarà comunque restituito 1 al suo posto.

In sostanza l'esecuzione di `longjmp` è analoga a quella di una istruzione `return`, solo che invece di ritornare alla riga successiva della funzione chiamante, il programma in questo caso ritorna alla posizione della relativa `setjmp`. L'altra differenza fondamentale con `return` è che il ritorno può essere effettuato anche attraverso diversi livelli di funzioni annidate.

L'implementazione di queste funzioni comporta alcune restrizioni dato che esse interagiscono direttamente con la gestione dello *stack* ed il funzionamento del compilatore stesso. In

³⁶a meno che, come precisa [?], alla chiusura di ciascuna fase non siano associate operazioni di pulizia specifiche (come deallocazioni, chiusure di file, ecc.), che non potrebbero essere eseguite con un salto non-locale.

³⁷anche questo è un classico esempio di variabile di *tipo opaco*.

particolare `setjmp` è implementata con una macro, pertanto non si può cercare di ottenerne l'indirizzo, ed inoltre le chiamate a questa funzione sono sicure solo in uno dei seguenti casi:

- come espressione di controllo in un comando condizionale, di selezione o di iterazione (come `if`, `switch` o `while`);
- come operando per un operatore di uguaglianza o confronto in una espressione di controllo di un comando condizionale, di selezione o di iterazione;
- come operando per l'operatore di negazione (!) in una espressione di controllo di un comando condizionale, di selezione o di iterazione;
- come espressione a sé stante.

In generale, dato che l'unica differenza fra il risultato di una chiamata diretta di `setjmp` e quello ottenuto nell'uscita con un `longjmp` è costituita dal valore di ritorno della funzione, quest'ultima viene usualmente chiamata all'interno di un'istruzione `if` che permetta di distinguere i due casi.

Uno dei punti critici dei salti non-locali è quello del valore delle variabili, ed in particolare quello delle variabili automatiche della funzione a cui si ritorna. In generale le variabili globali e statiche mantengono i valori che avevano al momento della chiamata di `longjmp`, ma quelli delle variabili automatiche (o di quelle dichiarate `register`) sono in genere indeterminati.

Quello che succede infatti è che i valori delle variabili che sono tenute in memoria manterranno il valore avuto al momento della chiamata di `longjmp`, mentre quelli tenuti nei registri del processore (che nella chiamata ad un'altra funzione vengono salvati nel contesto nello *stack*) torneranno al valore avuto al momento della chiamata di `setjmp`; per questo quando si vuole avere un comportamento coerente si può bloccare l'ottimizzazione che porta le variabili nei registri dichiarandole tutte come `volatile`.³⁸

2.4.4 La *endianness*

Un altro dei problemi di programmazione che può dar luogo ad effetti imprevisti è quello relativo alla cosiddetta *endianness*. Questa è una caratteristica generale dell'architettura hardware di un computer che dipende dal fatto che la rappresentazione di un numero binario può essere fatta in due modi, chiamati rispettivamente *big endian* e *little endian*, a seconda di come i singoli bit vengono aggregati per formare le variabili intere (ed in genere in diretta corrispondenza a come sono poi in realtà cablati sui bus interni del computer).

Figura 2.6: Schema della disposizione dei dati in memoria a seconda della *endianness*.

Per capire meglio il problema si consideri un intero a 32 bit scritto in una locazione di memoria posta ad un certo indirizzo. Come illustrato in fig. 2.6 i singoli bit possono essere disposti in memoria in due modi: a partire dal più significativo o a partire dal meno significativo. Così nel primo caso si troverà il byte che contiene i bit più significativi all'indirizzo menzionato e il byte con i bit meno significativi nell'indirizzo successivo; questo ordinamento è detto *big endian*, dato che si trova per prima la parte più grande. Il caso opposto, in cui si parte dal bit meno significativo è detto per lo stesso motivo *little endian*.

³⁸la direttiva `volatile` informa il compilatore che la variabile che è dichiarata può essere modificata, durante l'esecuzione del nostro, da altri programmi. Per questo motivo occorre dire al compilatore che non deve essere mai utilizzata l'ottimizzazione per cui quanto opportuno essa viene mantenuta in un registro, poiché in questo modo si perderebbero le eventuali modifiche fatte dagli altri programmi (che avvengono solo in una copia posta in memoria).

Si può allora verificare quale tipo di *endianness* usa il proprio computer con un programma elementare che si limita ad assegnare un valore ad una variabile per poi ristamparne il contenuto leggendolo un byte alla volta. Il codice di detto programma, `endtest.c`, è nei sorgenti allegati, allora se lo eseguiamo su un normale PC compatibile, che è *little endian* otterremo qualcosa del tipo:

```
[piccardi@gont sources]$ ./endtest
Using value ABCDEF01
val[0]= 1
val[1]=EF
val[2]=CD
val[3]=AB
```

mentre su un vecchio Macintosh con PowerPC, che è *big endian* avremo qualcosa del tipo:

```
piccardi@anarres:~/gopil/sources$ ./endtest
Using value ABCDEF01
val[0]=AB
val[1]=CD
val[2]=EF
val[3]= 1
```

L'attenzione alla *endianness* nella programmazione è importante, perché se si fanno assunzioni relative alla propria architettura non è detto che queste restino valide su un'altra architettura. Inoltre, come vedremo ad esempio in sez. 14.3, si possono avere problemi quando ci si trova a usare valori di un formato con una infrastruttura che ne usa un altro.

La *endianness* di un computer dipende essenzialmente dalla architettura hardware usata; Intel e Digital usano il *little endian*, Motorola, IBM, Sun (sostanzialmente tutti gli altri) usano il *big endian*. Il formato dei dati contenuti nelle intestazioni dei protocolli di rete (il cosiddetto *network order*) è anch'esso *big endian*; altri esempi di uso di questi due diversi formati sono quello del bus PCI, che è *little endian*, o quello del bus VME che è *big endian*.

Esistono poi anche dei processori che possono scegliere il tipo di formato all'avvio e alcuni che, come il PowerPC o l'Intel i860, possono pure passare da un tipo di ordinamento all'altro con una specifica istruzione. In ogni caso in Linux l'ordinamento è definito dall'architettura e dopo l'avvio del sistema in genere resta sempre lo stesso,³⁹ anche quando il processore permetterebbe di eseguire questi cambiamenti.

```
1 int endian(void)
2 {
3     /*
4      * Variables definition
5      */
6     short magic, test;
7     char * ptr;
8
9     magic = 0xABCD;                /* endianness magic number */
10    ptr = (char *) &magic;
11    test = (ptr[1]<<8) + (ptr[0]&0xFF); /* build value byte by byte */
12    return (magic == test);          /* if the same is little endian */
13 }
```

Figura 2.7: La funzione `endian`, usata per controllare il tipo di architettura della macchina.

³⁹su architettura PowerPC è possibile cambiarlo, si veda sez. 9.2.1.

Per controllare quale tipo di ordinamento si ha sul proprio computer si è scritta una piccola funzione di controllo, il cui codice è riportato fig. 2.7, che restituisce un valore nullo (falso) se l'architettura è *big endian* ed uno non nullo (vero) se l'architettura è *little endian*.

Come si vede la funzione è molto semplice, e si limita, una volta assegnato (9) un valore di test pari a `0xABCD` ad una variabile di tipo `short` (cioè a 16 bit), a ricostruirne una copia byte a byte. Per questo prima (10) si definisce il puntatore `ptr` per accedere al contenuto della prima variabile, ed infine calcola (11) il valore della seconda assumendo che il primo byte sia quello meno significativo (cioè, per quanto visto in fig. 2.6, che sia *little endian*). Infine la funzione restituisce (12) il valore del confronto delle due variabili.

In generale non ci si deve preoccupare della *endianness* all'interno di un programma fintanto che questo non deve generare o manipolare dei dati che sono scambiati con altre macchine, ad esempio via rete o tramite dei file binari. Nel primo caso la scelta è già stata fatta nella standardizzazione dei protocolli, che hanno adottato il *big endian* (che viene detto anche per questo *network order*); vedremo in sez. 14.3.1 le funzioni di conversione che devono essere usate.

Nel secondo caso occorre sapere quale *endianness* è stata usata nei dati memorizzati sul file e tenerne conto nella rilettura e nella manipolazione e relativa modifica (e salvataggio). La gran parte dei formati binari standardizzati specificano quale *endianness* viene utilizzata e basterà identificare qual'è, se se ne deve definire uno per i propri scopi basterà scegliere una volta per tutte quale usare e attenersi alla scelta.

Capitolo 3

La gestione dei processi

Come accennato nell'introduzione in un sistema unix-like tutte le operazioni vengono svolte tramite opportuni processi. In sostanza questi ultimi vengono a costituire l'unità base per l'allocazione e l'uso delle risorse del sistema.

Nel precedente capitolo abbiamo esaminato il funzionamento di un processo come unità a se stante, in questo esamineremo il funzionamento dei processi all'interno del sistema. Saranno cioè affrontati i dettagli della creazione e della terminazione dei processi, della gestione dei loro attributi e privilegi, e di tutte le funzioni a questo connesse. Infine nella sezione finale introdurremo alcune problematiche generiche della programmazione in ambiente *multitasking*.

3.1 Le funzioni di base della gestione dei processi

In questa sezione tratteremo le problematiche della gestione dei processi all'interno del sistema, illustrandone tutti i dettagli. Inizieremo con una panoramica dell'architettura dei processi, tratteremo poi le funzioni elementari che permettono di leggerne gli identificatori, per poi passare alla spiegazione delle funzioni base che si usano per la creazione e la terminazione dei processi, e per la messa in esecuzione degli altri programmi.

3.1.1 L'architettura della gestione dei processi

A differenza di quanto avviene in altri sistemi, ad esempio nel VMS, dove la generazione di nuovi processi è un'operazione privilegiata, una delle caratteristiche fondanti di Unix, che esamineremo in dettaglio più avanti, è che qualunque processo può a sua volta generarne altri. Ogni processo è identificato presso il sistema da un numero univoco, il cosiddetto *Process ID*, o più brevemente *PID*, assegnato in forma progressiva (vedi sez. 3.1.2) quando il processo viene creato.

Una seconda caratteristica di un sistema unix-like è che la generazione di un processo è un'operazione separata rispetto al lancio di un programma. In genere la sequenza è sempre quella di creare un nuovo processo, il quale eseguirà, in un passo successivo, il programma desiderato: questo è ad esempio quello che fa la shell quando mette in esecuzione il programma che gli indichiamo nella linea di comando.

Una terza caratteristica del sistema è che ogni processo è sempre stato generato da un altro processo, il processo generato viene chiamato *processo figlio* (*child process*) mentre quello che lo ha generato viene chiamato *processo padre* (*parent process*). Questo vale per tutti i processi, con una sola eccezione; dato che ci deve essere un punto di partenza esiste un processo iniziale

(che normalmente è `/sbin/init`), che come accennato in sez. 1.1.2 viene lanciato dal kernel alla conclusione della fase di avvio. Essendo questo il primo processo lanciato dal sistema ha sempre *PID* uguale a 1 e non è figlio di nessun altro processo.

Ovviamente `init` è un processo particolare che in genere si occupa di lanciare tutti gli altri processi necessari al funzionamento del sistema, inoltre `init` è essenziale per svolgere una serie di compiti amministrativi nelle operazioni ordinarie del sistema (torneremo su alcuni di essi in sez. 3.1.4) e non può mai essere terminato. La struttura del sistema comunque consente di lanciare al posto di `init` qualunque altro programma, e in casi di emergenza (ad esempio se il file di `init` si fosse corrotto) è ad esempio possibile lanciare una shell al suo posto.¹

```
[piccardi@gont piccardi]$ pstree -n
init--keventd
|-kpm-idled
|-kreiserfsd
|-portmap
|-syslogd
|-klogd
|-named
|-rpc.statd
|-gpm
|-inetd
|-junkbuster
|-master--qmgr
|   `--pickup
|-sshd
|-xfs
|-cron
|-bash---startx---xinit--XFree86
|   `--WindowMaker--ssh-agent
|       |-wtime
|       |-wmmon
|       |---wmmount
|       |---wmppp
|       |---wmcube
|       |---wmmixer
|       |---wmgtemp
|       |---wterm---bash---pstree
|       |---wterm---bash--emacs
|       |   `--man---pager
|   |-5*[getty]
|   |-snort
|   `--wwwoffled
```

Figura 3.1: L'albero dei processi, così come riportato dal comando `pstree`.

Dato che tutti i processi attivi nel sistema sono comunque generati da `init` o da uno dei suoi figli si possono classificare i processi con la relazione padre/figlio in un'organizzazione gerarchica ad albero. In fig. 3.1 si è mostrato il risultato del comando `pstree` che permette di visualizzare questa struttura, alla cui base c'è `init` che è progenitore di tutti gli altri processi.²

Il kernel mantiene una tabella dei processi attivi, la cosiddetta *process table*. Per ciascun processo viene mantenuta una voce in questa tabella, costituita da una struttura `task_struct`, che contiene tutte le informazioni rilevanti per quel processo. Tutte le strutture usate a questo

¹la cosa si fa passando la riga `init=/bin/sh` come parametro di avvio del kernel, l'argomento è di natura sistemistica e trattato in sez. 5.3 di [?].

²in realtà questo non è del tutto vero, in Linux, specialmente nelle versioni più recenti del kernel, ci sono alcuni processi speciali (come `keventd`, `kswapd`, ecc.) che pur comparendo nei comandi come figli di `init`, o con *PID* successivi ad uno, sono in realtà processi interni al kernel e che non rientrano in questa classificazione.

scopo sono dichiarate nell'*header file* `linux/sched.h`, ed in fig. 3.2 si è riportato uno schema semplificato che mostra la struttura delle principali informazioni contenute nella `task_struct`, che in seguito incontreremo a più riprese.

Figura 3.2: Schema semplificato dell'architettura delle strutture (`task_struct`, `fs_struct`, `file_struct`) usate dal kernel nella gestione dei processi.

Come accennato in sez. 1.1 è lo *scheduler* che decide quale processo mettere in esecuzione; esso viene eseguito in occasione di dell'invocazione di ogni *system call* ed per ogni interrupt dall'hardware oltre che in una serie di altre occasioni, e può essere anche attivato esplicitamente. Il timer di sistema provvede comunque a che esso sia invocato periodicamente, generando un interrupt periodico secondo una frequenza predeterminata, specificata dalla costante HZ del kernel (torneremo su questo argomento in sez. 6.4.1), che assicura che lo *scheduler* venga comunque eseguito ad intervalli regolari e possa prendere le sue decisioni.

A partire dal kernel 2.6.21 è stato introdotto anche un meccanismo completamente diverso, detto *tickless*, in cui non c'è più una interruzione periodica con frequenza prefissata, ma ad ogni chiamata del timer viene programmata l'interruzione successiva sulla base di una stima; in questo modo si evita di dover eseguire un migliaio di interruzioni al secondo anche su macchine che non stanno facendo nulla, con un forte risparmio nell'uso dell'energia da parte del processore che può essere messo in stato di sospensione anche per lunghi periodi di tempo.

Ma, indipendentemente dalle motivazioni per cui questo avviene, ogni volta che viene eseguito lo *scheduler* effettua il calcolo delle priorità dei vari processi attivi (torneremo su questo in sez. 3.3) e stabilisce quale di essi debba essere posto in esecuzione fino alla successiva invocazione.

3.1.2 Gli identificatori dei processi

Come accennato nella sezione precedente ogni processo viene identificato dal sistema da un numero identificativo univoco, il *process ID* o *PID*. Questo è un tipo di dato standard, `pid_t`, che in genere è un intero con segno (nel caso di Linux e della *glibc* il tipo usato è `int`).

Il *PID* viene assegnato in forma progressiva ogni volta che un nuovo processo viene creato,³ fino ad un limite che, essendo il tradizionalmente il *PID* un numero positivo memorizzato in un intero a 16 bit, arriva ad un massimo di 32768. Oltre questo valore l'assegnazione riparte dal numero più basso disponibile a partire da un minimo di 300,⁴ che serve a riservare i *PID* più bassi ai processi eseguiti direttamente dal kernel. Per questo motivo, come visto in sez. 3.1.1, il processo di avvio (`init`) ha sempre il *PID* uguale a uno.

Tutti i processi inoltre memorizzano anche il *PID* del genitore da cui sono stati creati, questo viene chiamato in genere *PPID* (da *Parent Process ID*). Questi due identificativi possono essere ottenuti usando le due funzioni di sistema `getpid` e `getppid`, i cui prototipi sono:

³in genere viene assegnato il numero successivo a quello usato per l'ultimo processo creato, a meno che questo numero non sia già utilizzato per un altro *PID*, *pgid* o *sid* (vedi sez. 8.1.2).

⁴questi valori, fino al kernel 2.4.x, erano definiti dalla macro `PID_MAX` nei file `threads.h` e `fork.c` dei sorgenti del kernel, con il 2.6.x e la nuova interfaccia per i *thread* anche il meccanismo di allocazione dei *PID* è stato modificato ed il valore massimo è impostabile attraverso il file `/proc/sys/kernel/pid_max` e di default vale 32768.

#include <sys/types.h>	
#include <unistd.h>	
pid_t getpid(void)	Restituisce il <i>PID</i> del processo corrente..
pid_t getppid(void)	Restituisce il <i>PID</i> del padre del processo corrente.

Entrambe le funzioni non riportano condizioni di errore.
--

esempi dell'uso di queste funzioni sono riportati in fig. 3.3, nel programma `fork_test.c`.

Il fatto che il *PID* sia un numero univoco per il sistema lo rende un candidato per generare ulteriori indicatori associati al processo di cui diventa possibile garantire l'unicità: ad esempio in alcune implementazioni la funzione `tempnam` (si veda sez. 4.2.6) usa il *PID* per generare un *pathname* univoco, che non potrà essere replicato da un altro processo che usi la stessa funzione. Questo utilizzo però può risultare pericoloso, un *PID* infatti è univoco solo fintanto che un processo è attivo, una volta terminato esso potrà essere riutilizzato da un processo completamente diverso, e di questo bisogna essere ben consapevoli.

Tutti i processi figli dello stesso processo padre sono detti *sibling*, questa è una delle relazioni usate nel *controllo di sessione*, in cui si raggruppano i processi creati su uno stesso terminale, o relativi allo stesso login. Torneremo su questo argomento in dettaglio in cap. 8, dove esamineremo gli altri identificativi associati ad un processo e le varie relazioni fra processi utilizzate per definire una sessione.

Oltre al *PID* e al *PPID*, e a quelli che vedremo in sez. 8.1.2, relativi al controllo di sessione, ad ogni processo vengono associati degli ulteriori identificatori ed in particolare quelli che vengono usati per il controllo di accesso. Questi servono per determinare se un processo può eseguire o meno le operazioni richieste, a seconda dei privilegi e dell'identità di chi lo ha posto in esecuzione; l'argomento è complesso e sarà affrontato in dettaglio in sez. 3.2.

3.1.3 La funzione `fork` e le funzioni di creazione dei processi

La funzione di sistema `fork` è la funzione fondamentale della gestione dei processi: come si è detto tradizionalmente l'unico modo di creare un nuovo processo era attraverso l'uso di questa funzione,⁵ essa quindi riveste un ruolo centrale tutte le volte che si devono scrivere programmi che usano il *multitasking*.⁶ Il prototipo di `fork` è:

#include <unistd.h>	
pid_t fork(void)	Crea un nuovo processo.

La funzione ritorna in caso di successo il <i>PID</i> del figlio nel padre e 0 nel figlio mentre ritorna -1 nel padre, senza creare il figlio, per un errore, al caso <code>errno</code> assumerà uno dei valori:	
EAGAIN	non ci sono risorse sufficienti per creare un altro processo (per allocare la tabella delle pagine e le strutture del task) o si è esaurito il numero di processi disponibili.
ENOMEM	non è stato possibile allocare la memoria per le strutture necessarie al kernel per creare il nuovo processo.

Dopo il successo dell'esecuzione di una `fork` sia il processo padre che il processo figlio continuano ad essere eseguiti normalmente, a partire dall'istruzione successiva alla `fork`. Il processo figlio è una copia del padre, e riceve una copia dei segmenti di testo, dati e dello *stack*

⁵in realtà oggi la *system call* usata da Linux per creare nuovi processi è `clone` (vedi 9.3.1), anche perché a partire dalla *glibc* 2.3.3 non viene più usata la *system call* originale, ma la stessa `fork` viene implementata tramite `clone`, cosa che consente una migliore interazione coi *thread*.

⁶oggi questa rilevanza, con la diffusione dell'uso dei *thread* che tratteremo al cap. 12, è in parte minore, ma `fork` resta comunque la funzione principale per la creazione di processi.

(vedi sez. 2.2.2), ed esegue esattamente lo stesso codice del padre. Si tenga presente però che la memoria è copiata e non condivisa, pertanto padre e figlio vedranno variabili diverse e le eventuali modifiche saranno totalmente indipendenti.

Per quanto riguarda la gestione della memoria, in generale il segmento di testo, che è identico per i due processi, è condiviso e tenuto in sola lettura per il padre e per i figli. Per gli altri segmenti Linux utilizza la tecnica del *copy on write*. Questa tecnica comporta che una pagina di memoria viene effettivamente copiata per il nuovo processo solo quando ci viene effettuata sopra una scrittura, e si ha quindi una reale differenza fra padre e figlio. In questo modo si rende molto più efficiente il meccanismo della creazione di un nuovo processo, non essendo più necessaria la copia di tutto lo spazio degli indirizzi virtuali del padre, ma solo delle pagine di memoria che sono state modificate, e solo al momento della modifica stessa.

La differenza che si ha nei due processi è che nel processo padre il valore di ritorno della funzione `fork` è il *PID* del processo figlio, mentre nel figlio è zero; in questo modo il programma può identificare se viene eseguito dal padre o dal figlio. Si noti come la funzione `fork` ritorni due volte, una nel padre e una nel figlio.

La scelta di questi valori di ritorno non è casuale, un processo infatti può avere più figli, ed il valore di ritorno di `fork` è l'unico che gli permette di identificare qual è quello appena creato. Al contrario un figlio ha sempre un solo padre il cui *PID*, come spiegato in sez. 3.1.2, può sempre essere ottenuto con `getppid`; per questo si ritorna un valore nullo, che non è il *PID* di nessun processo.

Normalmente la chiamata a `fork` può fallire solo per due ragioni: o ci sono già troppi processi nel sistema, il che di solito è sintomo che qualcos'altro non sta andando per il verso giusto, o si è ecceduto il limite sul numero totale di processi permessi all'utente, argomento che tratteremo in dettaglio in sez. 6.3.2.

L'uso di `fork` avviene secondo due modalità principali; la prima è quella in cui all'interno di un programma si creano processi figli cui viene affidata l'esecuzione di una certa sezione di codice, mentre il processo padre ne esegue un'altra. È il caso tipico dei programmi server (il modello *client-server* è illustrato in sez. 13.1.1) in cui il padre riceve ed accetta le richieste da parte dei programmi client, per ciascuna delle quali pone in esecuzione un figlio che è incaricato di fornire le risposte associate al servizio.

La seconda modalità è quella in cui il processo vuole eseguire un altro programma; questo è ad esempio il caso della shell. In questo caso il processo crea un figlio la cui unica operazione è quella di fare una `exec` (di cui parleremo in sez. 3.1.6) subito dopo la `fork`.

Alcuni sistemi operativi (il VMS ad esempio) combinano le operazioni di questa seconda modalità (una `fork` seguita da una `exec`) in un'unica operazione che viene chiamata *spawn*. Nei sistemi unix-like è stato scelto di mantenere questa separazione, dato che, come per la prima modalità d'uso, esistono numerosi scenari in cui si può usare una `fork` senza aver bisogno di eseguire una `exec`.

Inoltre, anche nel caso della seconda modalità d'uso, avere le due funzioni separate permette al figlio di cambiare alcune caratteristiche del processo (maschera dei segnali, redirectione dell'output, utente per conto del cui viene eseguito, e molto altro su cui torneremo in seguito) prima della `exec`, rendendo così relativamente facile intervenire sulle le modalità di esecuzione del nuovo programma.

In fig. 3.3 è riportato il corpo del codice del programma di esempio `forktest`, che permette di illustrare molte caratteristiche dell'uso della funzione `fork`. Il programma crea un numero di figli specificato da linea di comando, e prende anche alcune opzioni per indicare degli eventuali tempi di attesa in secondi (eseguiti tramite la funzione `sleep`) per il padre ed il figlio (con `forktest -h` si ottiene la descrizione delle opzioni). Il codice completo, compresa la parte che

```

1 #include <errno.h>           /* error definitions and routines */
2 #include <stdlib.h>          /* C standard library */
3 #include <unistd.h>          /* unix standard library */
4 #include <stdio.h>           /* standard I/O library */
5 #include <string.h>          /* string functions */
6
7 /* Help printing routine */
8 void usage(void);
9
10 int main(int argc, char *argv[])
11 {
12     /*
13      * Variables definition
14      */
15     int nchild, i;
16     pid_t pid;
17     int wait_child = 0;
18     int wait_parent = 0;
19     int wait_end = 0;
20     ... /* handling options */
21     nchild = atoi(argv[optind]);
22     printf("Test_for_forking_%d_child\n", nchild);
23     /* loop to fork children */
24     for (i=0; i<nchild; i++) {
25         if ( (pid = fork()) < 0) {
26             /* on error exit */
27             printf("Error_on_%d_child_creation,_%s\n", i+1, strerror(errno));
28             exit(-1);
29         }
30         if (pid == 0) { /* child */
31             printf("Child_%d_successfully_executing\n", ++i);
32             if (wait_child) sleep(wait_child);
33             printf("Child_%d,_parent_%d,_exiting\n", i, getppid());
34             exit(0);
35         } else { /* parent */
36             printf("Spawned_%d_child,_pid_%d\n", i+1, pid);
37             if (wait_parent) sleep(wait_parent);
38             printf("Go_to_next_child\n");
39         }
40     }
41     /* normal exit */
42     if (wait_end) sleep(wait_end);
43     return 0;
44 }

```

Figura 3.3: Esempio di codice per la creazione di nuovi processi (da `fork_test.c`).

gestisce le opzioni a riga di comando, è disponibile nel file `fork_test.c`, distribuito insieme agli altri sorgenti degli esempi della guida su <http://gail.gnulinux.it>.

Decifrato il numero di figli da creare, il ciclo principale del programma (24-40) esegue in successione la creazione dei processi figli controllando il successo della chiamata a `fork` (25-29); ciascun figlio (31-34) si limita a stampare il suo numero di successione, eventualmente attendere il numero di secondi specificato e scrivere un messaggio prima di uscire. Il processo padre invece (36-38) stampa un messaggio di creazione, eventualmente attende il numero di secondi specificato, e procede nell'esecuzione del ciclo; alla conclusione del ciclo, prima di uscire, può essere specificato un altro periodo di attesa.

Se eseguiamo il comando, che è preceduto dall'istruzione `export LD_LIBRARY_PATH=./` per permettere l'uso delle librerie dinamiche, senza specificare attese (come si può notare in (17-19) i valori predefiniti specificano di non attendere), otterremo come risultato sul terminale:

```
[piccardi@selidor sources]$ export LD_LIBRARY_PATH=./; ./forktest 3
Process 1963: forking 3 child
Spawned 1 child, pid 1964
Child 1 successfully executing
Child 1, parent 1963, exiting
Go to next child
Spawned 2 child, pid 1965
Child 2 successfully executing
Child 2, parent 1963, exiting
Go to next child
Child 3 successfully executing
Child 3, parent 1963, exiting
Spawned 3 child, pid 1966
Go to next child
```

Esaminiamo questo risultato: una prima conclusione che si può trarre è che non si può dire quale processo fra il padre ed il figlio venga eseguito per primo dopo la chiamata a `fork`; dall'esempio si può notare infatti come nei primi due cicli sia stato eseguito per primo il padre (con la stampa del *PID* del nuovo processo) per poi passare all'esecuzione del figlio (completata con i due avvisi di esecuzione ed uscita), e tornare all'esecuzione del padre (con la stampa del passaggio al ciclo successivo), mentre la terza volta è stato prima eseguito il figlio (fino alla conclusione) e poi il padre.

In generale l'ordine di esecuzione dipenderà, oltre che dall'algoritmo di *scheduling* usato dal kernel, dalla particolare situazione in cui si trova la macchina al momento della chiamata, risultando del tutto imprevedibile. Eseguendo più volte il programma di prova e producendo un numero diverso di figli, si sono ottenute situazioni completamente diverse, compreso il caso in cui il processo padre ha eseguito più di una `fork` prima che uno dei figli venisse messo in esecuzione.

Pertanto non si può fare nessuna assunzione sulla sequenza di esecuzione delle istruzioni del codice fra padre e figli, né sull'ordine in cui questi potranno essere messi in esecuzione. Se è necessaria una qualche forma di precedenza occorrerà provvedere ad espliciti meccanismi di sincronizzazione, pena il rischio di incorrere nelle cosiddette *race condition* (vedi sez. 3.4.2).

In realtà con l'introduzione dei kernel della serie 2.6 lo *scheduler* è stato modificato per eseguire sempre per primo il figlio.⁷ Questa è una ottimizzazione adottata per evitare che il padre, effettuando per primo una operazione di scrittura in memoria, attivasse il meccanismo del *copy on write*, operazione inutile quando il figlio viene creato solo per eseguire una `exec` per lanciare un altro programma che scarta completamente lo spazio degli indirizzi e rende superflua la copia della memoria modificata dal padre. Eseguendo sempre per primo il figlio la `exec` verrebbe effettuata subito, con la certezza di utilizzare il *copy on write* solo quando necessario.

Con il kernel 2.6.32 però il comportamento è stato nuovamente cambiato, stavolta facendo eseguire per primo sempre il padre. Si è realizzato infatti che l'eventualità prospettata per la scelta precedente era comunque poco probabile, mentre l'esecuzione immediata del padre presenta sempre il vantaggio di poter utilizzare immediatamente tutti i dati che sono nella cache della CPU e nell'unità di gestione della memoria virtuale, senza doverli invalidare, cosa che per i processori moderni, che hanno linee di cache interne molto profonde, avrebbe un forte impatto sulle prestazioni.

⁷i risultati precedenti infatti sono stati ottenuti usando un kernel della serie 2.4.

Allora anche se quanto detto in precedenza si verifica nel comportamento effettivo dei programmi soltanto per i kernel fino alla serie 2.4, per mantenere la portabilità con altri kernel unix-like e con i diversi comportamenti adottati dalle Linux nella sua evoluzione, è comunque opportuno non fare nessuna assunzione sull'ordine di esecuzione di padre e figlio dopo la chiamata a `fork`.

Si noti infine come dopo la `fork`, essendo i segmenti di memoria utilizzati dai singoli processi completamente indipendenti, le modifiche delle variabili nei processi figli, come l'incremento di `i` in (31), sono visibili solo a loro, (ogni processo vede solo la propria copia della memoria), e non hanno alcun effetto sul valore che le stesse variabili hanno nel processo padre ed in eventuali altri processi figli che eseguano lo stesso codice.

Un secondo aspetto molto importante nella creazione dei processi figli è quello dell'interazione dei vari processi con i file. Ne parleremo qui anche se buona parte dei concetti relativi ai file verranno trattati più avanti (principalmente in sez. 5.1). Per illustrare meglio quello che avviene si può reindirizzare su un file l'output del programma di test, quello che otterremo è:

```
[piccardi@selidor sources]$ ./forktest 3 > output
[piccardi@selidor sources]$ cat output
Process 1967: forking 3 child
Child 1 successfully executing
Child 1, parent 1967, exiting
Test for forking 3 child
Spawned 1 child, pid 1968
Go to next child
Child 2 successfully executing
Child 2, parent 1967, exiting
Test for forking 3 child
Spawned 1 child, pid 1968
Go to next child
Spawned 2 child, pid 1969
Go to next child
Child 3 successfully executing
Child 3, parent 1967, exiting
Test for forking 3 child
Spawned 1 child, pid 1968
Go to next child
Spawned 2 child, pid 1969
Go to next child
Spawned 3 child, pid 1970
Go to next child
```

che come si vede è completamente diverso da quanto ottenevamo sul terminale.

Il comportamento delle varie funzioni di interfaccia con i file è analizzato in gran dettaglio in sez. 5.1 per l'interfaccia nativa Unix ed in sez. 5.3 per la standardizzazione adottata nelle librerie del linguaggio C, valida per qualunque sistema operativo.

Qui basta accennare che si sono usate le funzioni standard della libreria del C che prevedono l'output bufferizzato. Il punto è che questa bufferizzazione (che tratteremo in dettaglio in sez. 5.3.2) varia a seconda che si tratti di un file su disco, in cui il buffer viene scaricato su disco solo quando necessario, o di un terminale, in cui il buffer viene scaricato ad ogni carattere di "a capo".

Nel primo esempio allora avevamo che, essendovi un a capo nella stringa stampata, ad ogni chiamata a `printf` il buffer veniva scaricato, per cui le singole righe comparivano a video subito dopo l'esecuzione della `printf`. Ma con la redirezione su file la scrittura non avviene più alla fine di ogni riga e l'output resta nel buffer.

Dato che ogni figlio riceve una copia della memoria del padre, esso riceverà anche quanto c'è nel buffer delle funzioni di I/O, comprese le linee scritte dal padre fino allora. Così quando

il buffer viene scritto su disco all'uscita del figlio, troveremo nel file anche tutto quello che il processo padre aveva scritto prima della sua creazione. E alla fine del file (dato che in questo caso il padre esce per ultimo) troveremo anche l'output completo del padre.

L'esempio ci mostra un altro aspetto fondamentale dell'interazione con i file, valido anche per l'esempio precedente, ma meno evidente: il fatto cioè che non solo processi diversi possono scrivere in contemporanea sullo stesso file (l'argomento dell'accesso concorrente ai file è trattato in dettaglio in sez. 5.2.1), ma anche che, a differenza di quanto avviene per le variabili in memoria, la posizione corrente sul file è condivisa fra il padre e tutti i processi figli.

Quello che succede è che quando lo *standard output*⁸ del padre viene rediretto come si è fatto nell'esempio, lo stesso avviene anche per tutti i figli. La funzione *fork* infatti ha la caratteristica di duplicare nei processi figli tutti i *file descriptor* (vedi sez. 5.1.1) dei file aperti nel processo padre (allo stesso modo in cui lo fa la funzione *dup*, trattata in sez. 5.2.2). Ciò fa sì che padre e figli condividano le stesse voci della *file table* (tratteremo in dettaglio questi termini in sez. 5.1.1 e sez. 5.2.1) fra le quali c'è anche la posizione corrente nel file.

Quando un processo scrive su un file la posizione corrente viene aggiornata sulla *file table*, e tutti gli altri processi, che vedono la stessa *file table*, vedranno il nuovo valore. In questo modo si evita, in casi come quello appena mostrato in cui diversi figli scrivono sullo stesso file usato dal padre, che una scrittura eseguita in un secondo tempo da un processo vada a sovrapporsi a quelle precedenti: l'output potrà risultare mescolato, ma non ci saranno parti perdute per via di una sovrascrittura.

Questo tipo di comportamento è essenziale in tutti quei casi in cui il padre crea un figlio e attende la sua conclusione per proseguire, ed entrambi scrivono sullo stesso file. Un caso tipico di questo comportamento è la shell quando lancia un programma. In questo modo, anche se lo standard output viene rediretto, il padre potrà sempre continuare a scrivere in coda a quanto scritto dal figlio in maniera automatica; se così non fosse ottenere questo comportamento sarebbe estremamente complesso, necessitando di una qualche forma di comunicazione fra i due processi per far riprendere al padre la scrittura al punto giusto.

In generale comunque non è buona norma far scrivere più processi sullo stesso file senza una qualche forma di sincronizzazione in quanto, come visto anche con il nostro esempio, le varie scritture risulteranno mescolate fra loro in una sequenza imprevedibile. Per questo le modalità con cui in genere si usano i file dopo una *fork* sono sostanzialmente due:

1. Il processo padre aspetta la conclusione del figlio. In questo caso non è necessaria nessuna azione riguardo ai file, in quanto la sincronizzazione della posizione corrente dopo eventuali operazioni di lettura e scrittura effettuate dal figlio è automatica.
2. L'esecuzione di padre e figlio procede indipendentemente. In questo caso ciascuno dei due processi deve chiudere i file che non gli servono una volta che la *fork* è stata eseguita, per evitare ogni forma di interferenza.

Oltre ai file aperti i processi figli ereditano dal padre una serie di altre proprietà; la lista dettagliata delle proprietà che padre e figlio hanno in comune dopo l'esecuzione di una *fork* è la seguente:

- i file aperti e gli eventuali flag di *close-on-exec* impostati (vedi sez. 3.1.6 e sez. 5.2.1);
- gli identificatori per il controllo di accesso: l'*user-ID* reale, il *group-ID* reale, l'*user-ID* effettivo, il *group-ID* effettivo ed i *group-ID* supplementari (vedi sez. 3.2.1);
- gli identificatori per il controllo di sessione: il *process group-ID* e il *session id* ed il terminale di controllo (vedi sez. 8.1.2);

⁸si chiama così il file su cui di default un programma scrive i suoi dati in uscita, tratteremo l'argomento in dettaglio in sez. 5.1.1.

- la directory di lavoro (vedi sez. 4.2.4) e la directory radice (vedi sez. 4.5.4);
- la maschera dei permessi di creazione dei file (vedi sez. 4.4.3);
- la maschera dei segnali bloccati (vedi sez. 7.4.4) e le azioni installate (vedi sez. 7.3.1);
- i segmenti di memoria condivisa agganciati al processo (vedi sez. 11.2.6);
- i limiti sulle risorse (vedi sez. 6.3.2);
- il valori di *nice*, le priorità *real-time* e le affinità di processore (vedi sez. 3.3.2, sez. 3.3.3 e sez. 3.3.4);
- le variabili di ambiente (vedi sez. 2.3.3).
- l'insieme dei descrittori associati alle code di messaggi POSIX (vedi sez. 11.4.2) che vengono copiate come i *file descriptor*, questo significa che entrambi condivideranno gli stessi flag.

Oltre a quelle relative ad un diverso spazio degli indirizzi (e una memoria totalmente indipendente) le differenze fra padre e figlio dopo l'esecuzione di una `fork` invece sono:⁹

- il valore di ritorno di `fork`;
- il *PID* (*process id*), quello del figlio viene assegnato ad un nuovo valore univoco;
- il *PPID* (*parent process id*), quello del figlio viene impostato al *PID* del padre;
- i valori dei tempi di esecuzione (vedi sez. 6.4.2) e delle risorse usate (vedi sez. 6.3.1), che nel figlio sono posti a zero;
- i *lock* sui file (vedi sez. 10.1) e sulla memoria (vedi sez. 2.2.4), che non vengono ereditati dal figlio;
- gli allarmi, i timer (vedi sez. 7.3.4) ed i segnali pendenti (vedi sez. 7.3.1), che per il figlio vengono cancellati.
- le operazioni di I/O asincrono in corso (vedi sez. 10.3.3) che non vengono ereditate dal figlio;
- gli aggiustamenti fatti dal padre ai semafori con `semop` (vedi sez. 11.2.5).
- le notifiche sui cambiamenti delle directory con *dnotify* (vedi sez. 7.1.4), che non vengono ereditate dal figlio;
- le mappature di memoria marcate come `MADV_DONTFORK` (vedi sez. 10.4.1) che non vengono ereditate dal figlio;
- l'impostazione con `prctl` (vedi sez. 9.2.1) che notifica al figlio la terminazione del padre viene cancellata se presente nel padre;
- il segnale di terminazione del figlio è sempre `SIGCHLD` anche qualora nel padre fosse stato modificato (vedi sez. 9.3.1).

Una seconda funzione storica usata per la creazione di un nuovo processo è `vfork`, che è esattamente identica a `fork` ed ha la stessa semantica e gli stessi errori; la sola differenza è che non viene creata la tabella delle pagine né la struttura dei task per il nuovo processo. Il processo padre è posto in attesa fintanto che il figlio non ha eseguito una `execve` o non è uscito con una `_exit`. Il figlio condivide la memoria del padre (e modifiche possono avere effetti imprevedibili) e non deve ritornare o uscire con `exit` ma usare esplicitamente `_exit`.

Questa funzione è un rimasuglio dei vecchi tempi in cui eseguire una `fork` comportava anche la copia completa del segmento dati del processo padre, che costituiva un inutile appesantimento in tutti quei casi in cui la `fork` veniva fatta solo per poi eseguire una `exec`. La funzione venne introdotta in BSD per migliorare le prestazioni.

Dato che Linux supporta il *copy on write* la perdita di prestazioni è assolutamente trascurabile, e l'uso di questa funzione, che resta un caso speciale della *system call clone*

⁹a parte le ultime quattro, relative a funzionalità specifiche di Linux, le altre sono esplicitamente menzionate dallo standard POSIX.1-2001.

(che tratteremo in dettaglio in sez. 9.3.1) è deprecato; per questo eviteremo di trattarla ulteriormente.

3.1.4 La conclusione di un processo

In sez. 2.1.3 abbiamo già affrontato le modalità con cui chiudere un programma, ma dall'interno del programma stesso. Avendo a che fare con un sistema *multitasking* resta da affrontare l'argomento dal punto di vista di come il sistema gestisce la conclusione dei processi.

Abbiamo visto in sez. 2.1.3 le tre modalità con cui un programma viene terminato in maniera normale: la chiamata di `exit`, che esegue le funzioni registrate per l'uscita e chiude gli *stream* e poi esegue `_exit`, il ritorno dalla funzione `main` equivalente alla chiamata di `exit`, e la chiamata diretta a `_exit`, che passa direttamente alle operazioni di terminazione del processo da parte del kernel.

Ma abbiamo accennato che oltre alla conclusione normale esistono anche delle modalità di conclusione anomala. Queste sono in sostanza due: il programma può chiamare la funzione `abort` (vedi sez. 7.3.4) per invocare una chiusura anomala, o essere terminato da un segnale (torneremo sui segnali in cap. 7). In realtà anche la prima modalità si riconduce alla seconda, dato che `abort` si limita a generare il segnale `SIGABRT`.

Qualunque sia la modalità di conclusione di un processo, il kernel esegue comunque una serie di operazioni di terminazione: chiude tutti i file aperti, rilascia la memoria che stava usando, e così via; l'elenco completo delle operazioni eseguite alla chiusura di un processo è il seguente:

- tutti i *file descriptor* (vedi sez. 5.1.1) sono chiusi;
- viene memorizzato lo stato di terminazione del processo;
- ad ogni processo figlio viene assegnato un nuovo padre (in genere `init`);
- viene inviato il segnale `SIGCHLD` al processo padre (vedi sez. 7.3.6);
- se il processo è un leader di sessione ed il suo terminale di controllo è quello della sessione viene mandato un segnale di `SIGHUP` a tutti i processi del gruppo di *foreground* e il terminale di controllo viene disconnesso (vedi sez. 8.1.3);
- se la conclusione di un processo rende orfano un *process group* ciascun membro del gruppo viene bloccato, e poi gli vengono inviati in successione i segnali `SIGHUP` e `SIGCONT` (vedi ancora sez. 8.1.3).

Oltre queste operazioni è però necessario poter disporre di un meccanismo ulteriore che consenta di sapere come la terminazione è avvenuta: dato che in un sistema unix-like tutto viene gestito attraverso i processi, il meccanismo scelto consiste nel riportare lo *stato di terminazione* (il cosiddetto *termination status*) al processo padre.

Nel caso di conclusione normale, abbiamo visto in sez. 2.1.3 che lo stato di uscita del processo viene caratterizzato tramite il valore del cosiddetto *exit status*, cioè il valore passato come argomento alle funzioni `exit` o `_exit` o il valore di ritorno per `main`. Ma se il processo viene concluso in maniera anomala il programma non può specificare nessun *exit status*, ed è il kernel che deve generare autonomamente il *termination status* per indicare le ragioni della conclusione anomala.

Si noti la distinzione fra *exit status* e *termination status*: quello che contraddistingue lo stato di chiusura del processo e viene riportato attraverso le funzioni `wait` o `waitpid` (vedi sez. 3.1.5) è sempre quest'ultimo; in caso di conclusione normale il kernel usa il primo (nel codice eseguito da `_exit`) per produrre il secondo.

La scelta di riportare al padre lo stato di terminazione dei figli, pur essendo l'unica possibile, comporta comunque alcune complicazioni: infatti se alla sua creazione è scontato che

ogni nuovo processo abbia un padre, non è detto che sia così alla sua conclusione, dato che il padre potrebbe essere già terminato; si potrebbe avere cioè quello che si chiama un processo *orfano*.

Questa complicazione viene superata facendo in modo che il processo orfano venga *adottato* da *init*, o meglio dal processo con *PID* 1,¹⁰ cioè quello lanciato direttamente dal kernel all'avvio, che sta alla base dell'albero dei processi visto in sez. 3.1.1 e che anche per questo motivo ha un ruolo essenziale nel sistema e non può mai terminare.¹¹

Come già accennato quando un processo termina, il kernel controlla se è il padre di altri processi in esecuzione: in caso positivo allora il *PPID* di tutti questi processi verrà sostituito dal kernel con il *PID* di *init*, cioè con 1. In questo modo ogni processo avrà sempre un padre (nel caso possiamo parlare di un padre *adottivo*) cui riportare il suo stato di terminazione.

Come verifica di questo comportamento possiamo eseguire il nostro programma *forktest* imponendo a ciascun processo figlio due secondi di attesa prima di uscire, il risultato è:

```
[piccardi@selidor sources]$ ./forktest -c2 3
Process 1972: forking 3 child
Spawned 1 child, pid 1973
Child 1 successfully executing
Go to next child
Spawned 2 child, pid 1974
Child 2 successfully executing
Go to next child
Child 3 successfully executing
Spawned 3 child, pid 1975
Go to next child

[piccardi@selidor sources]$ Child 3, parent 1, exiting
Child 2, parent 1, exiting
Child 1, parent 1, exiting
```

come si può notare in questo caso il processo padre si conclude prima dei figli, tornando alla shell, che stampa il prompt sul terminale: circa due secondi dopo viene stampato a video anche l'output dei tre figli che terminano, e come si può notare in questo caso, al contrario di quanto visto in precedenza, essi riportano 1 come *PPID*.

Altrettanto rilevante è il caso in cui il figlio termina prima del padre, perché non è detto che il padre sia in esecuzione e possa ricevere immediatamente lo stato di terminazione, quindi il kernel deve comunque conservare una certa quantità di informazioni riguardo ai processi che sta terminando.

Questo viene fatto mantenendo attiva la voce nella tabella dei processi, e memorizzando alcuni dati essenziali, come il *PID*, i tempi di CPU usati dal processo (vedi sez. 6.4.1) e lo stato di terminazione, mentre la memoria in uso ed i file aperti vengono rilasciati immediatamente.

I processi che sono terminati, ma il cui stato di terminazione non è stato ancora ricevuto dal padre sono chiamati *zombie*, essi restano presenti nella tabella dei processi ed in genere possono essere identificati dall'output di *ps* per la presenza di una Z nella colonna che ne indica lo stato (vedi tab. 3.8). Quando il padre effettuerà la lettura dello stato di terminazione anche questa informazione, non più necessaria, verrà scartata ed il processo potrà considerarsi completamente concluso.

Possiamo utilizzare il nostro programma di prova per analizzare anche questa condizione: lanciamo il comando *forktest* in *background* (vedi sez. 8.1), indicando al processo padre di

¹⁰anche se, come vedremo in sez. 9.2.1, a partire dal kernel 3.4 è diventato possibile delegare questo compito anche ad un altro processo.

¹¹almeno non senza un blocco completo del sistema, in caso di terminazione o di non esecuzione di *init* infatti il kernel si blocca con un cosiddetto *kernel panic*, dato che questo è un errore fatale.

aspettare 10 secondi prima di uscire. In questo caso, usando `ps` sullo stesso terminale (prima dello scadere dei 10 secondi) otterremo:

```
[piccardi@selidor sources]$ ps T
PID TTY      STAT   TIME COMMAND
 419 pts/0    S       0:00 bash
 568 pts/0    S       0:00 ./forktest -e10 3
 569 pts/0    Z       0:00 [forktest <defunct>]
 570 pts/0    Z       0:00 [forktest <defunct>]
 571 pts/0    Z       0:00 [forktest <defunct>]
 572 pts/0    R       0:00 ps T
```

e come si vede, dato che non si è fatto nulla per riceverne lo stato di terminazione, i tre processi figli sono ancora presenti pur essendosi conclusi, con lo stato di *zombie* e l'indicazione che sono terminati (la scritta *defunct*).

La possibilità di avere degli *zombie* deve essere tenuta sempre presente quando si scrive un programma che deve essere mantenuto in esecuzione a lungo e creare molti processi figli. In questo caso si deve sempre avere cura di far leggere al programma l'eventuale stato di uscita di tutti i figli. Una modalità comune di farlo è attraverso l'utilizzo di un apposito *signal handler* che chiami la funzione `wait`, (vedi sez. 3.1.5), ne esamineremo in dettaglio un esempio (fig. 7.4) in sez. 7.3.6.

La lettura degli stati di uscita è necessaria perché anche se gli *zombie* non consumano risorse di memoria o processore, occupano comunque una voce nella tabella dei processi e se li si lasciano accumulare a lungo quest'ultima potrebbe esaurirsi, con la conseguente impossibilità di lanciare nuovi processi.

Si noti tuttavia che quando un processo adottato da `init` termina, non diviene mai uno *zombie*. Questo perché una delle funzioni di `init` è appunto quella di chiamare la funzione `wait` per i processi a cui fa da padre, completandone la terminazione. Questo è quanto avviene anche quando, come nel caso del precedente esempio con `forktest`, il padre termina con dei figli in stato di *zombie*. Questi scompaiono quando, alla terminazione del padre dopo i secondi programmati, tutti figli che avevamo generato, e che erano diventati *zombie*, vengono adottati da `init`, il quale provvede a completarne la terminazione.

Si tenga presente infine che siccome gli *zombie* sono processi già terminati, non c'è modo di eliminarli con il comando `kill` o inviandogli un qualunque segnale di terminazione (l'argomento è trattato in sez. 7.2.3). Qualora ci si trovi in questa situazione l'unica possibilità di cancellarli dalla tabella dei processi è quella di terminare il processo che li ha generati e che non sta facendo il suo lavoro, in modo che `init` possa adottarli e concluderne correttamente la terminazione.

Si tenga anche presente che la presenza di *zombie* nella tabella dei processi non è sempre indice di un qualche malfunzionamento, in una macchina con molto carico infatti può esservi una presenza temporanea dovuta al fatto che il processo padre ancora non ha avuto il tempo di gestirli.

3.1.5 Le funzioni di attesa e ricezione degli stati di uscita

Uno degli usi più comuni delle capacità *multitasking* di un sistema unix-like consiste nella creazione di programmi di tipo server, in cui un processo principale attende le richieste che vengono poi soddisfatte da una serie di processi figli.

Si è già sottolineato al paragrafo precedente come in questo caso diventi necessario gestire esplicitamente la conclusione dei figli onde evitare di riempire di *zombie* la tabella dei processi. Tratteremo in questa sezione le funzioni di sistema deputate a questo compito; la prima è `wait` ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
```

Attende la terminazione di un processo.

La funzione ritorna il *PID* del figlio in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

ECHILD il processo non ha nessun figlio di cui attendere uno stato di terminazione.

EINTR la funzione è stata interrotta da un segnale.

Questa funzione è presente fin dalle prime versioni di Unix ed è quella usata tradizionalmente per attendere la terminazione dei figli. La funzione sospende l'esecuzione del processo corrente e ritorna non appena un qualunque processo figlio termina. Se un figlio è già terminato prima della sua chiamata la funzione ritorna immediatamente, se più processi figli sono già terminati occorrerà continuare a chiamare la funzione più volte fintanto che non si è recuperato lo stato di terminazione di tutti quanti.

Al ritorno della funzione lo stato di terminazione del figlio viene salvato (come *value result argument*) nella variabile puntata da *status* e tutte le risorse del kernel relative al processo (vedi sez. 3.1.4) vengono rilasciate. Nel caso un processo abbia più figli il valore di ritorno della funzione sarà impostato al *PID* del processo di cui si è ricevuto lo stato di terminazione, cosa che permette di identificare qual è il figlio che è terminato.

Questa funzione ha il difetto di essere poco flessibile, in quanto ritorna all'uscita di un qualunque processo figlio. Nelle occasioni in cui è necessario attendere la conclusione di uno specifico processo fra tutti quelli esistenti occorre predisporre un meccanismo che tenga conto di tutti processi che sono terminati, e provveda a ripetere la chiamata alla funzione nel caso il processo cercato non risulti fra questi. Se infatti il processo cercato è già terminato e se è già ricevuto lo stato di uscita senza registrarlo, la funzione non ha modo di accorgersene, e si continuerà a chiamarla senza accorgersi che quanto interessava è già accaduto.

Per questo motivo lo standard POSIX.1 ha introdotto una seconda funzione che effettua lo stesso servizio, ma dispone di una serie di funzionalità più ampie, legate anche al controllo di sessione (si veda sez. 8.1). Dato che è possibile ottenere lo stesso comportamento di *wait*¹² si consiglia di utilizzare sempre questa nuova funzione di sistema, *waitpid*, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options)
```

Attende il cambiamento di stato di un processo figlio.

La funzione ritorna il *PID* del processo che ha cambiato stato in caso di successo, o 0 se è stata specificata l'opzione *WNOHANG* e il processo non è uscito e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

ECHILD il processo specificato da *pid* non esiste o non è figlio del processo chiamante.

EINTR non è stata specificata l'opzione *WNOHANG* e la funzione è stata interrotta da un segnale.

EINVAL si è specificato un valore non valido per l'argomento *options*.

La prima differenza fra le due funzioni è che con *waitpid* si può specificare in maniera flessibile quale processo attendere, sulla base del valore fornito dall'argomento *pid*. Questo può assumere diversi valori, secondo lo specchietto riportato in tab. 3.1, dove si sono riportate anche le costanti definite per indicare alcuni di essi.

Il comportamento di *waitpid* può inoltre essere modificato passando alla funzione delle opportune opzioni tramite l'argomento *options*; questo deve essere specificato come maschera

¹²in effetti il codice *wait(&status)* è del tutto equivalente a *waitpid(WAIT_ANY, &status, 0)*.

Valore	Costante	Significato
< -1	–	Attende per un figlio il cui <i>process group</i> (vedi sez. 8.1.2) è uguale al valore assoluto di <i>pid</i> .
-1	WAIT_ANY	Attende per un figlio qualsiasi, usata in questa maniera senza specificare nessuna opzione è equivalente a <i>wait</i> .
0	WAIT_MYPGRP	Attende per un figlio il cui <i>process group</i> (vedi sez. 8.1.2) è uguale a quello del processo chiamante.
> 0	–	Attende per un figlio il cui <i>PID</i> è uguale al valore di <i>pid</i> .

Tabella 3.1: Significato dei valori dell'argomento *pid* della funzione *waitpid*.

binaria delle costanti riportati nella prima parte in tab. 3.2 che possono essere combinate fra loro con un OR aritmetico. Nella seconda parte della stessa tabella si sono riportati anche alcune opzioni non standard specifiche di Linux, che consentono un controllo più dettagliato per i processi creati con la *system call* generica *clone* (vedi sez. 9.3.1) e che vengono usati principalmente per la gestione della terminazione dei *thread* (vedi sez. ??).

Costante	Descrizione
WNOHANG	La funzione ritorna immediatamente anche se non è terminato nessun processo figlio.
WUNTRACED	Ritorna anche quando un processo figlio è stato fermato.
WCONTINUED	Ritorna anche quando un processo figlio che era stato fermato ha ripreso l'esecuzione (dal kernel 2.6.10).
__WCLONE	Attende solo per i figli creati con <i>clone</i> (vedi sez. 9.3.1), vale a dire processi che non emettono nessun segnale o emettono un segnale diverso da SIGCHLD alla terminazione, il default è attendere soltanto i processi figli ordinari ignorando quelli creati da <i>clone</i> .
__WALL	Attende per qualunque figlio, sia ordinario che creato con <i>clone</i> , se specificata con __WCLONE quest'ultima viene ignorata.
__WNOTHREAD	Non attende per i figli di altri <i>thread</i> dello stesso <i>thread group</i> , questo era il comportamento di default del kernel 2.4 che non supportava la possibilità, divenuta il default a partire dal 2.6, di attendere un qualunque figlio appartenente allo stesso <i>thread group</i> .

Tabella 3.2: Costanti che identificano i bit dell'argomento *options* della funzione *waitpid*.

L'uso dell'opzione WNOHANG consente di prevenire il blocco della funzione qualora nessun figlio sia uscito o non si siano verificate le altre condizioni per l'uscita della funzione. In tal caso. In tal caso la funzione, invece di restituire il *PID* del processo (che è sempre un intero positivo) ritornerà un valore nullo.

Le altre due opzioni, WUNTRACED e WCONTINUED, consentono rispettivamente di tracciare non la terminazione di un processo, ma il fatto che esso sia stato fermato, o fatto ripartire, e sono utilizzate per la gestione del controllo di sessione (vedi sez. 8.1).

Nel caso di WUNTRACED la funzione ritorna, restituendone il *PID*, quando un processo figlio entra nello stato *stopped*¹³ (vedi tab. 3.8), mentre con WCONTINUED la funzione ritorna quando un processo in stato *stopped* riprende l'esecuzione per la ricezione del segnale SIGCONT (l'uso di questi segnali per il controllo di sessione è trattato in sez. 8.1.3).

La terminazione di un processo figlio (così come gli altri eventi osservabili con *waitpid*) è chiaramente un evento asincrono rispetto all'esecuzione di un programma e può avvenire in un qualunque momento. Per questo motivo, come accennato nella sezione precedente, una

¹³in realtà viene notificato soltanto il caso in cui il processo è stato fermato da un segnale di stop (vedi sez. 8.1.3), e non quello in cui lo stato *stopped* è dovuto all'uso di *ptrace* (vedi sez. 9.2.2).

delle azioni prese dal kernel alla conclusione di un processo è quella di mandare un segnale di SIGCHLD al padre. L'azione predefinita per questo segnale (si veda sez. 7.1.1) è di essere ignorato, ma la sua generazione costituisce il meccanismo di comunicazione asincrona con cui il kernel avverte il processo padre che uno dei suoi figli è terminato.

Il comportamento delle funzioni è però cambiato nel passaggio dal kernel 2.4 al kernel 2.6, quest'ultimo infatti si è adeguato alle prescrizioni dello standard POSIX.1-2001 e come da esso richiesto se SIGCHLD viene ignorato, o se si imposta il flag di SA_NOCLDSTOP nella ricezione dello stesso (si veda sez. 7.4.3) i processi figli che terminano non diventano *zombie* e sia `wait` che `waitpid` si bloccano fintanto che tutti i processi figli non sono terminati, dopo di che falliscono con un errore di ENOCHLD.¹⁴

Con i kernel della serie 2.4 e tutti i kernel delle serie precedenti entrambe le funzioni di attesa ignorano questa prescrizione e si comportano sempre nello stesso modo,¹⁵ indipendentemente dal fatto SIGCHLD sia ignorato o meno: attendono la terminazione di un processo figlio e ritornano il relativo *PID* e lo stato di terminazione nell'argomento `status`.

In generale in un programma non si vuole essere forzati ad attendere la conclusione di un processo figlio per proseguire l'esecuzione, specie se tutto questo serve solo per leggerne lo stato di chiusura (ed evitare eventualmente la presenza di *zombie*). Per questo la modalità più comune di chiamare queste funzioni è quella di utilizzarle all'interno di un *signal handler* (vedremo un esempio di come gestire SIGCHLD con i segnali in sez. 7.4.1). In questo caso infatti, dato che il segnale è generato dalla terminazione di un figlio, avremo la certezza che la chiamata a `waitpid` non si bloccherà.

Come accennato sia `wait` che `waitpid` restituiscono lo stato di terminazione del processo tramite il puntatore `status`, e se non interessa memorizzarlo si può passare un puntatore nullo. Il valore restituito da entrambe le funzioni dipende dall'implementazione, ma tradizionalmente gli 8 bit meno significativi sono riservati per memorizzare lo stato di uscita del processo, e gli altri per indicare il segnale che ha causato la terminazione (in caso di conclusione anomala), uno per indicare se è stato generato un *core dump* (vedi sez. 7.2.1), ecc.¹⁶

Lo standard POSIX.1 definisce una serie di macro di preprocessore da usare per analizzare lo stato di uscita. Esse sono definite sempre in `<sys/wait.h>` ed elencate in tab. 3.3. Si tenga presente che queste macro prevedono che gli si passi come parametro la variabile di tipo `int` puntata dall'argomento `status` restituito da `wait` o `waitpid`.

Si tenga conto che nel caso di conclusione anomala il valore restituito da `WTERMSIG` può essere confrontato con le costanti che identificano i segnali definite in `signal.h` ed elencate in tab. 7.1, e stampato usando le apposite funzioni trattate in sez. 7.2.9.

A partire dal kernel 2.6.9, sempre in conformità allo standard POSIX.1-2001, è stata introdotta una nuova funzione di attesa che consente di avere un controllo molto più preciso sui possibili cambiamenti di stato dei processi figli e più dettagli sullo stato di uscita; la funzione di sistema è `waitid` ed il suo prototipo è:

¹⁴questo è anche il motivo per cui le opzioni WUNTRACED e WCONTINUED sono utilizzabili soltanto qualora non si sia impostato il flag di SA_NOCLDSTOP per il segnale SIGCHLD.

¹⁵lo standard POSIX.1 originale infatti lascia indefinito il comportamento di queste funzioni quando SIGCHLD viene ignorato.

¹⁶le definizioni esatte si possono trovare in `<bits/waitstatus.h>` ma questo file non deve mai essere usato direttamente, esso viene incluso attraverso `<sys/wait.h>`.

¹⁷questa macro non è definita dallo standard POSIX.1-2001, ma è presente come estensione sia in Linux che in altri Unix, deve essere pertanto utilizzata con attenzione (ad esempio è il caso di usarla in un blocco `#ifdef WCOREDUMP ... #endif`).

Macro	Descrizione
WIFEXITED(s)	Condizione vera (valore non nullo) per un processo figlio che sia terminato normalmente.
WEXITSTATUS(s)	Restituisce gli otto bit meno significativi dello stato di uscita del processo (passato attraverso <code>_exit</code> , <code>exit</code> o come valore di ritorno di <code>main</code>); può essere valutata solo se <code>WIFEXITED</code> ha restituito un valore non nullo.
WIFSIGNALED(s)	Condizione vera se il processo figlio è terminato in maniera anomala a causa di un segnale che non è stato catturato (vedi sez. 7.1.4).
WTERMSIG(s)	Restituisce il numero del segnale che ha causato la terminazione anomala del processo; può essere valutata solo se <code>WIFSIGNALED</code> ha restituito un valore non nullo.
WCOREDUMP(s)	Vera se il processo terminato ha generato un file di <i>core dump</i> ; può essere valutata solo se <code>WIFSIGNALED</code> ha restituito un valore non nullo. ¹⁷
WIFSTOPPED(s)	Vera se il processo che ha causato il ritorno di <code>waitpid</code> è bloccato; l'uso è possibile solo con <code>waitpid</code> avendo specificato l'opzione <code>WUNTRACED</code> .
WSTOPSIG(s)	Restituisce il numero del segnale che ha bloccato il processo; può essere valutata solo se <code>WIFSTOPPED</code> ha restituito un valore non nullo.
WIFCONTINUED(s)	Vera se il processo che ha causato il ritorno è stato riavviato da un <code>SIGCONT</code> (disponibile solo a partire dal kernel 2.6.10).

Tabella 3.3: Descrizione delle varie macro di preprocessore utilizzabili per verificare lo stato di terminazione s di un processo.

<pre>#include <sys/types.h> #include <sys/wait.h> int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)</pre>	
	Attende il cambiamento di stato di un processo figlio.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
ECHILD	il processo specificato da <code>pid</code> non esiste o non è figlio del processo chiamante.
EINTR	non è stata specificata l'opzione <code>WNOHANG</code> e la funzione è stata interrotta da un segnale.
EINVAL	si è specificato un valore non valido per l'argomento <code>options</code> .

La funzione prevede che si specifichi quali processi si intendono osservare usando i due argomenti `idtype` ed `id`; il primo indica se ci si vuole porre in attesa su un singolo processo, un gruppo di processi o un processo qualsiasi, e deve essere specificato secondo uno dei valori di tab. 3.4; il secondo indica, a seconda del valore del primo, quale processo o quale gruppo di processi selezionare.

Valore	Descrizione
P_PID	Indica la richiesta di attendere per un processo figlio il cui <i>PID</i> corrisponda al valore dell'argomento <code>id</code> .
P_PGID	Indica la richiesta di attendere per un processo figlio appartenente al <i>process group</i> (vedi sez. 8.1.2) il cui <i>pgid</i> corrisponda al valore dell'argomento <code>id</code> .
P_ALL	Indica la richiesta di attendere per un processo figlio generico, il valore dell'argomento <code>id</code> viene ignorato.

Tabella 3.4: Costanti per i valori dell'argomento `idtype` della funzione `waitid`.

Come per `waitpid` anche il comportamento di `waitid` è controllato dall'argomento `options`, da specificare come maschera binaria dei valori riportati in tab. 3.5. Benché alcuni di questi siano identici come significato ed effetto ai precedenti di tab. 3.2, ci sono delle differenze significative: in questo caso si dovrà specificare esplicitamente l'attesa della terminazione di un processo impostando l'opzione `WEXITED`, mentre il precedente `WUNTRACED` è sostituito

da WSTOPPED. Infine è stata aggiunta l'opzione WNOWAIT che consente una lettura dello stato mantenendo il processo in attesa di ricezione, così che una successiva chiamata possa di nuovo riceverne lo stato.

Valore	Descrizione
WEXITED	Ritorna quando un processo figlio è terminato.
WNOHANG	Ritorna immediatamente anche se non c'è niente da notificare.
WSTOPPED	Ritorna quando un processo figlio è stato fermato.
WCONTINUED	Ritorna quando un processo figlio che era stato fermato ha ripreso l'esecuzione.
WNOWAIT	Lascia il processo ancora in attesa di ricezione, così che una successiva chiamata possa di nuovo riceverne lo stato.

Tabella 3.5: Costanti che identificano i bit dell'argomento options della funzione waitid.

La funzione waitid restituisce un valore nullo in caso di successo, e -1 in caso di errore; viene restituito un valore nullo anche se è stata specificata l'opzione WNOHANG e la funzione è ritornata immediatamente senza che nessun figlio sia terminato. Pertanto per verificare il motivo del ritorno della funzione occorre analizzare le informazioni che essa restituisce; queste, al contrario delle precedenti wait e waitpid che usavano un semplice valore numerico, sono ritornate in una struttura di tipo siginfo_t (vedi fig. 7.9) all'indirizzo puntato dall'argomento infop.

Tratteremo nei dettagli la struttura siginfo_t ed il significato dei suoi vari campi in sez. 7.4.3, per quanto ci interessa qui basta dire che al ritorno di waitid verranno avvalorati i seguenti campi:

- si_pid con il *PID* del figlio.
- si_uid con l'*user-ID* reale (vedi sez. 3.2) del figlio.
- si_signo con SIGCHLD.
- si_status con lo stato di uscita del figlio o con il segnale che lo ha terminato, fermato o riavviato.
- si_code con uno fra CLD_EXITED, CLD_KILLED, CLD_STOPPED, CLD_CONTINUED, CLD_TRAPPED e CLD_DUMPED a indicare la ragione del ritorno della funzione, il cui significato è, nell'ordine: uscita normale, terminazione da segnale, processo fermato, processo riavviato, processo terminato in *core dump* (vedi sez. 7.2.1).

Infine Linux, seguendo un'estensione di BSD, supporta altre due funzioni per la lettura dello stato di terminazione di un processo, analoghe alle precedenti ma che prevedono un ulteriore argomento attraverso il quale il kernel può restituire al padre informazioni sulle risorse (vedi sez. 6.3) usate dal processo terminato e dai vari figli. Le due funzioni di sistema sono wait3 e wait4, che diventano accessibili definendo la macro _USE_BSD, i loro prototipi sono:

```
#include <sys/types.h>
#include <sys/times.h>
#include <sys/resource.h>
#include <sys/wait.h>
int wait3(int *status, int options, struct rusage *rusage)
int wait4(pid_t pid, int *status, int options, struct rusage *rusage)
    Attende il cambiamento di stato di un processo figlio, riportando l'uso delle risorse.
```

La funzione ha gli stessi valori di ritorno e codici di errore di waitpid.

La funzione `wait4` è identica `waitpid` sia nel comportamento che per i valori dei primi tre argomenti, ma in più restituisce nell'argomento aggiuntivo `rusage` un sommario delle risorse usate dal processo. Questo argomento è una struttura di tipo `rusage` definita in `sys/resource.h`, che viene utilizzata anche dalla funzione `getrusage` per ottenere le risorse di sistema usate da un processo. La sua definizione è riportata in fig. 6.6 e ne tratteremo in dettaglio il significato sez. 6.3.1. La funzione `wait3` è semplicemente un caso particolare di (e con Linux viene realizzata con la stessa *system call*), ed è equivalente a chiamare `wait4(-1, &status, opt, rusage)`, per questo motivo è ormai deprecata in favore di `wait4`.

3.1.6 La famiglia delle funzioni `exec` per l'esecuzione dei programmi

Abbiamo già detto che una delle modalità principali con cui si utilizzano i processi in Unix è quella di usarli per lanciare nuovi programmi: questo viene fatto attraverso una delle funzioni della famiglia `exec`. Quando un processo chiama una di queste funzioni esso viene completamente sostituito dal nuovo programma, il *PID* del processo non cambia, dato che non viene creato un nuovo processo, la funzione semplicemente rimpiazza lo *stack*, i dati ed il testo del processo corrente con un nuovo programma letto da disco, eseguendo il *link-loader* con gli effetti illustrati in sez. 2.1.1.

Figura 3.4: La interrelazione fra le sei funzioni della famiglia `exec`.

Ci sono sei diverse versioni di `exec` (per questo la si è chiamata famiglia di funzioni) che possono essere usate per questo compito, in realtà (come mostrato in fig. 3.4), tutte queste funzioni sono tutte varianti che consentono di invocare in modi diversi, semplificando il passaggio degli argomenti, la funzione di sistema `execve`, il cui prototipo è:

```
#include <unistd.h>
int execve(const char *filename, char *const argv[], char *const envp[])
                                                    Esegue un programma.
```

La funzione ritorna solo in caso di errore, restituendo `-1`, nel qual caso `errno` assumerà uno dei valori:

EACCES	il file o l'interprete non file ordinari, o non sono eseguibili, o il file è su un filesystem montato con l'opzione <code>noexec</code> , o manca il permesso di attraversamento di una delle directory del <i>pathname</i> .
EAGAIN	dopo un cambio di <i>UID</i> si è ancora sopra il numero massimo di processi consentiti per l'utente (dal kernel 3.1, per i dettagli vedi sez. 3.2.2).
EINVAL	l'eseguibile ELF ha più di un segmento <code>PT_INTERP</code> , cioè chiede di essere eseguito da più di un interprete.
ELIBBAD	un interprete ELF non è in un formato riconoscibile.
ENOENT	il file o una delle librerie dinamiche o l'interprete necessari per eseguirlo non esistono.
ENOEXEC	il file è in un formato non eseguibile o non riconosciuto come tale, o compilato per un'altra architettura.
EPERM	il file ha i bit <i>suid</i> o <i>sgid</i> e l'utente non è root, ed il processo viene tracciato, oppure il filesystem è montato con l'opzione <code>nosuid</code> .
ETXTBSY	l'eseguibile è aperto in scrittura da uno o più processi.
E2BIG	la lista degli argomenti è troppo grande.

ed inoltre `EFAULT`, `EIO`, `EISDIR`, `ELOOP`, `EMFILE`, `ENAMETOOLONG`, `ENFILE`, `ENOMEM`, `ENOTDIR` nel loro significato generico.

La funzione `execve` esegue il programma o lo script indicato dal *pathname* `filename`, passandogli la lista di argomenti indicata da `argv` e come ambiente la lista di stringhe indicata da `envp`. Entrambe le liste devono essere terminate da un puntatore nullo. I vettori degli argomenti e dell'ambiente possono essere acceduti dal nuovo programma quando la sua funzione `main` è dichiarata nella forma `main(int argc, char *argv[], char *envp[])`. Si tenga presente per il passaggio degli argomenti e dell'ambiente esistono comunque dei limiti, su cui torneremo in sez. 6.3).

In caso di successo la funzione non ritorna, in quanto al posto del programma chiamante viene eseguito il nuovo programma indicato da `filename`. Se il processo corrente è tracciato con `ptrace` (vedi sez. 9.2.2) in caso di successo viene emesso il segnale `SIGTRAP`.

Le altre funzioni della famiglia (`execl`, `execv`, `execle`, `execlp`, `execvp`) servono per fornire all'utente una serie di possibili diverse interfacce nelle modalità di passaggio degli argomenti all'esecuzione del nuovo programma. I loro prototipi sono:

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...)
int execv(const char *path, char *const argv[])
int execle(const char *path, const char *arg, ..., char * const envp[])
int execlp(const char *file, const char *arg, ...)
int execvp(const char *file, char *const argv[])

Eseguono un programma.
```

Le funzioni ritornano solo in caso di errore, restituendo `-1`, i codici di errore sono gli stessi di `execve`.

Tutte le funzioni mettono in esecuzione nel processo corrente il programma indicati nel primo argomento. Gli argomenti successivi consentono di specificare gli argomenti e l'ambiente che saranno ricevuti dal nuovo processo. Per capire meglio le differenze fra le funzioni della famiglia si può fare riferimento allo specchio riportato in tab. 3.6. La relazione fra le funzioni è invece illustrata in fig. 3.4.

Caratteristiche	Funzioni					
	execl	execlp	execle	execv	execvp	execve
argomenti a lista	•	•	•			
argomenti a vettore				•	•	•
filename completo	•		•	•		•
ricerca su PATH		•			•	
ambiente a vettore			•			•
uso di environ	•	•		•	•	

Tabella 3.6: Confronto delle caratteristiche delle varie funzioni della famiglia `exec`.

La prima differenza fra le funzioni riguarda le modalità di passaggio dei valori che poi andranno a costituire gli argomenti a linea di comando (cioè i valori di `argv` e `argc` visti dalla funzione `main` del programma chiamato). Queste modalità sono due e sono riassunte dagli mnemonici “v” e “l” che stanno rispettivamente per *vector* e *list*.

Nel primo caso gli argomenti sono passati tramite il vettore di puntatori `argv[]` a stringhe terminate con zero che costituiranno gli argomenti a riga di comando, questo vettore *deve* essere terminato da un puntatore nullo. Nel secondo caso le stringhe degli argomenti sono passate alla funzione come lista di puntatori, nella forma:

```
char *arg0, char *arg1, ..., char *argn, NULL
```

che deve essere terminata da un puntatore nullo. In entrambi i casi vale la convenzione che il primo argomento (`arg0` o `argv[0]`) viene usato per indicare il nome del file che contiene il programma che verrà eseguito.

La seconda differenza fra le funzioni riguarda le modalità con cui si specifica il programma che si vuole eseguire. Con lo mnemonico “p” si indicano le due funzioni che replicano il comportamento della shell nello specificare il comando da eseguire; quando l’argomento *file* non contiene una “/” esso viene considerato come un nome di programma, e viene eseguita automaticamente una ricerca fra i file presenti nella lista di directory specificate dalla variabile di ambiente *PATH*. Il file che viene posto in esecuzione è il primo che viene trovato. Se si ha un errore relativo a permessi di accesso insufficienti (cioè l’esecuzione della sottostante *execve* ritorna un *EACCES*), la ricerca viene proseguita nelle eventuali ulteriori directory indicate in *PATH*; solo se non viene trovato nessun altro file viene finalmente restituito *EACCES*. Le altre quattro funzioni si limitano invece a cercare di eseguire il file indicato dall’argomento *path*, che viene interpretato come il *pathname* del programma.

La terza differenza è come viene passata la lista delle variabili di ambiente. Con lo mnemonico “e” vengono indicate quelle funzioni che necessitano di un vettore di parametri *envp[]* analogo a quello usato per gli argomenti a riga di comando (terminato quindi da un *NULL*), le altre usano il valore della variabile *environ* (vedi sez. 2.3.3) del processo di partenza per costruire l’ambiente.

Oltre a mantenere lo stesso *PID*, il nuovo programma fatto partire da una delle funzioni della famiglia *exec* mantiene la gran parte delle proprietà del processo chiamante; una lista delle più significative è la seguente:

- il *process id* (*PID*) ed il *parent process id* (*PPID*);
- l’*user-ID* reale, il *group-ID* reale ed i *group-ID* supplementari (vedi sez. 3.2.1);
- la directory radice (vedi sez. 4.5.4) e la directory di lavoro corrente (vedi sez. 4.2.4);
- la maschera di creazione dei file (*umask*, vedi sez. 4.4.3) ed i *lock* sui file (vedi sez. 10.1);
- il valori di *nice*, le priorità real-time e le affinità di processore (vedi sez. 3.3.2; sez. 3.3.3 e sez. 3.3.4);
- il *session ID* (*sid*) ed il *process group ID* (*pgid*), vedi sez. 8.1.2;
- il terminale di controllo (vedi sez. 8.1.3);
- il tempo restante ad un allarme (vedi sez. 7.3.4);
- i limiti sulle risorse (vedi sez. 6.3.2);
- i valori delle variabili *tms_utime*, *tms_stime*; *tms_cutime*, *tms_ustime* (vedi sez. 6.4.2);
- la maschera dei segnali (si veda sez. 7.4.4);
- l’insieme dei segnali pendenti (vedi sez. 7.3.1).

Una serie di proprietà del processo originale, che non avrebbe senso mantenere in un programma che esegue un codice completamente diverso in uno spazio di indirizzi totalmente indipendente e ricreato da zero, vengono perse con l’esecuzione di una *exec*. Lo standard POSIX.1-2001 prevede che le seguenti proprietà non vengano preservate:

- gli eventuali stack alternativi per i segnali (vedi sez. 7.5.3);
- i *directory stream* (vedi sez. 4.2.3), che vengono chiusi;
- le mappature dei file in memoria (vedi sez. 10.4.1);
- i segmenti di memoria condivisa SysV (vedi sez. 11.2.6) e POSIX (vedi sez. 11.4.3);
- i *memory lock* (vedi sez. 2.2.4);
- le funzioni registrate all’uscita (vedi sez. 2.1.4);
- i semafori e le code di messaggi POSIX (vedi sez. 11.4.4 e sez. 11.4.2);
- i timer POSIX (vedi sez. 7.5.2).

Inoltre i segnali che sono stati impostati per essere ignorati nel processo chiamante mantengono la stessa impostazione pure nel nuovo programma, ma tutti gli altri segnali, ed in particolare quelli per i quali è stato installato un gestore vengono impostati alla loro azione

predefinita (vedi sez. 7.3.1). Un caso speciale è il segnale SIGCHLD che, quando impostato a SIG_IGN, potrebbe anche essere reimpostato a SIG_DFL. Lo standard POSIX.1-2001 prevede che questo comportamento sia deciso dalla singola implementazione, quella di Linux è di non modificare l'impostazione precedente.

Oltre alle precedenti, che sono completamente generali e disponibili anche su altri sistemi unix-like, esistono altre proprietà dei processi, attinenti alle caratteristiche specifiche di Linux, che non vengono preservate nell'esecuzione della funzione `exec`, queste sono:

- le operazioni di I/O asincrono (vedi sez. 10.3.3) pendenti vengono cancellate;
- le *capabilities* vengono modificate come illustrato in sez. 9.1.1;
- tutti i *thread* tranne il chiamante (vedi sez. ??) vengono cancellati e tutti gli oggetti ad essi relativi (vedi sez. ??) sono rimossi;
- viene impostato il flag PR_SET_DUMPABLE di `prctl` (vedi sez. 9.2.1) a meno che il programma da eseguire non sia *suid* o *sgid* (vedi sez. 3.2.1 e sez. 4.4.2);
- il flag PR_SET_KEEPCAPS di `prctl` (vedi sez. 9.2.1) viene cancellato;
- il nome del processo viene impostato al nome del file contenente il programma messo in esecuzione;
- il segnale di terminazione viene reimpostato a SIGCHLD;
- l'ambiente viene reinizializzato impostando le variabili attinenti alla localizzazione al valore di default POSIX.

La gestione dei file aperti nel passaggio al nuovo programma lanciato con `exec` dipende dal valore che ha il flag di *close-on-exec* per ciascun *file descriptor* (vedi sez. 5.2.1). I file per cui è impostato vengono chiusi, tutti gli altri file restano aperti. Questo significa che il comportamento predefinito è che i file restano aperti attraverso una `exec`, a meno di non aver impostato esplicitamente (in apertura o con `fnctl`) il suddetto flag. Per le directory, lo standard POSIX.1 richiede che esse vengano chiuse attraverso una `exec`, in genere questo è fatto dalla funzione `opendir` (vedi sez. 4.2.3) che effettua da sola l'impostazione del flag di *close-on-exec* sulle directory che apre, in maniera trasparente all'utente.

Il comportamento della funzione in relazione agli identificatori relativi al controllo di accesso verrà trattato in dettaglio in sez. 3.2, qui è sufficiente anticipare (si faccia riferimento a sez. 3.2.1 per la definizione di questi identificatori) come l'*user-ID reale* ed il *group-ID reale* restano sempre gli stessi, mentre l'*user-ID salvato* ed il *group-ID salvato* vengono impostati rispettivamente all'*user-ID effettivo* ed il *group-ID effettivo*. Questi ultimi normalmente non vengono modificati, a meno che il file di cui viene chiesta l'esecuzione non abbia o il *suid* bit o lo *sgid* bit impostato (vedi sez. 4.4.2), in questo caso l'*user-ID effettivo* ed il *group-ID effettivo* vengono impostati rispettivamente all'utente o al gruppo cui il file appartiene.

Se il file da eseguire è in formato *a.out* e necessita di librerie condivise, viene lanciato il *linker* dinamico `/lib/ld.so` prima del programma per caricare le librerie necessarie ed effettuare il link dell'eseguibile; il formato è ormai in completo disuso, per cui è molto probabile che non il relativo supporto non sia disponibile. Se il programma è in formato ELF per caricare le librerie dinamiche viene usato l'interprete indicato nel segmento PT_INTERP previsto dal formato stesso, in genere questo è `/lib/ld-linux.so.1` per programmi collegati con la *libc5*, e `/lib/ld-linux.so.2` per programmi collegati con la *glibc*.

Infine nel caso il programma che si vuole eseguire sia uno script e non un binario, questo deve essere un file di testo che deve iniziare con una linea nella forma:

```
#!/path/to/interpreter [argomenti]
```

dove l'interprete indicato deve essere un eseguibile binario e non un altro script, che verrà chiamato come se si fosse eseguito il comando `interpreter [argomenti] filename`.

Si tenga presente che con Linux quanto viene scritto come **argomenti** viene passato all'interprete come un unico argomento con una unica stringa di lunghezza massima di 127 caratteri e se questa dimensione viene ecceduta la stringa viene troncata; altri Unix hanno dimensioni massime diverse, e diversi comportamenti, ad esempio FreeBSD esegue la scansione della riga e la divide nei vari argomenti e se è troppo lunga restituisce un errore di ENAMETOOLONG; una comparazione dei vari comportamenti sui diversi sistemi unix-like si trova su <http://www.in-ulm.de/~mascheck/various/shebang/>.

Con la famiglia delle **exec** si chiude il novero delle funzioni su cui è basata la gestione tradizionale dei processi in Unix: con **fork** si crea un nuovo processo, con **exec** si lancia un nuovo programma, con **exit** e **wait** si effettua e verifica la conclusione dei processi. Tutte le altre funzioni sono ausiliarie e servono per la lettura e l'impostazione dei vari parametri connessi ai processi.

3.2 Il controllo di accesso

In questa sezione esamineremo le problematiche relative al controllo di accesso dal punto di vista dei processi; vedremo quali sono gli identificatori usati, come questi possono essere modificati nella creazione e nel lancio di nuovi processi, le varie funzioni per la loro manipolazione diretta e tutte le problematiche connesse ad una gestione accorta dei privilegi.

3.2.1 Gli identificatori del controllo di accesso

Come accennato in sez. 1.1.4 il modello base¹⁸ di sicurezza di un sistema unix-like è fondato sui concetti di utente e gruppo, e sulla separazione fra l'amministratore (*root*, detto spesso anche *superuser*) che non è sottoposto a restrizioni, ed il resto degli utenti, per i quali invece vengono effettuati i vari controlli di accesso.

Abbiamo già accennato come il sistema associ ad ogni utente e gruppo due identificatori univoci, lo *User-ID* (abbreviato in *UID*) ed il *Group-ID* (abbreviato in *GID*). Questi servono al kernel per identificare uno specifico utente o un gruppo di utenti, per poi poter controllare che essi siano autorizzati a compiere le operazioni richieste. Ad esempio in sez. 4.4 vedremo come ad ogni file vengano associati un utente ed un gruppo (i suoi *proprietari*, indicati appunto tramite un *UID* ed un *GID*) che vengono controllati dal kernel nella gestione dei permessi di accesso.

Dato che tutte le operazioni del sistema vengono compiute dai processi, è evidente che per poter implementare un controllo sulle operazioni occorre anche poter identificare chi è che ha lanciato un certo programma, e pertanto anche a ciascun processo dovrà essere associato un utente e un gruppo.

Un semplice controllo di una corrispondenza fra identificativi non garantisce però sufficiente flessibilità per tutti quei casi in cui è necessario poter disporre di privilegi diversi, o dover impersonare un altro utente per un limitato insieme di operazioni. Per questo motivo in generale tutti i sistemi unix-like prevedono che i processi abbiano almeno due gruppi di identificatori, chiamati rispettivamente *real* ed *effective* (cioè *reali* ed *effettivi*). Nel caso di

¹⁸in realtà già esistono estensioni di questo modello base, che lo rendono più flessibile e controllabile, come le *capabilities* illustrate in sez. 9.1.1, le ACL per i file (vedi sez. 4.5.2) o il *Mandatory Access Control* di *SELinux*; inoltre basandosi sul lavoro effettuato con *SELinux*, a partire dal kernel 2.5.x, è iniziato lo sviluppo di una infrastruttura di sicurezza, i *Linux Security Modules*, o LSM, in grado di fornire diversi agganci a livello del kernel per modularizzare tutti i possibili controlli di accesso, cosa che ha permesso di realizzare diverse alternative a *SELinux*.

Linux si aggiungono poi altri due gruppi, il *saved* (salvati) ed il *filesystem* (di filesystem), secondo la situazione illustrata in tab. 3.7.

Suffisso	Gruppo	Denominazione	Significato
uid gid	<i>real</i> "	<i>user-ID reale</i> <i>group-ID reale</i>	Indica l'utente che ha lanciato il programma. Indica il gruppo principale dell'utente che ha lanciato il programma.
euid egid —	<i>effective</i> " —	<i>user-ID effettivo</i> <i>group-ID effettivo</i> <i>group-ID supplementari</i>	Indica l'utente usato nel controllo di accesso. Indica il gruppo usato nel controllo di accesso. Indicano gli ulteriori gruppi cui l'utente appartiene.
— —	<i>saved</i> "	<i>user-ID salvato</i> <i>group-ID salvato</i>	Mantiene una copia dell'euid iniziale. Mantiene una copia dell'egid iniziale.
fsuid fsgid	<i>filesystem</i> "	<i>user-ID di filesystem</i> <i>group-ID di filesystem</i>	Indica l'utente effettivo per l'accesso al filesystem. Indica il gruppo effettivo per l'accesso al filesystem.

Tabella 3.7: Identificatori di utente e gruppo associati a ciascun processo con indicazione dei suffissi usati dalle varie funzioni di manipolazione.

Al primo gruppo appartengono l'*UID reale* ed il *GID reale*: questi vengono impostati al login ai valori corrispondenti all'utente con cui si accede al sistema (e relativo gruppo principale). Servono per l'identificazione dell'utente e normalmente non vengono mai cambiati. In realtà vedremo (in sez. 3.2.2) che è possibile modificarli, ma solo ad un processo che abbia i privilegi di amministratore; questa possibilità è usata proprio dal programma *login* che, una volta completata la procedura di autenticazione, lancia una shell per la quale imposta questi identificatori ai valori corrispondenti all'utente che entra nel sistema.

Al secondo gruppo appartengono l'*UID effettivo* e il *GID effettivo*, a cui si aggiungono gli eventuali *GID supplementari* dei gruppi dei quali l'utente fa parte. Questi sono invece gli identificatori usati nelle verifiche dei permessi del processo e per il controllo di accesso ai file (argomento affrontato in dettaglio in sez. 4.4.1).

Questi identificatori normalmente sono identici ai corrispondenti del gruppo *real* tranne nel caso in cui, come accennato in sez. 3.1.6, il programma che si è posto in esecuzione abbia i bit *suid* o *sgid* impostati (il significato di questi bit è affrontato in dettaglio in sez. 4.4.2). In questo caso essi saranno impostati all'utente e al gruppo proprietari del file. Questo consente, per programmi in cui ci sia questa necessità, di dare a qualunque utente i privilegi o i permessi di un altro, compreso l'amministratore.

Come nel caso del *PID* e del *PPID*, anche tutti questi identificatori possono essere ottenuti da un programma attraverso altrettante funzioni di sistema dedicate alla loro lettura, queste sono *getuid*, *geteuid*, *getgid* e *getegid*, ed i loro prototipi sono:

#include <unistd.h> #include <sys/types.h> uid_t getuid(void)	Legge l' <i>UID</i> reale del processo corrente.
uid_t geteuid(void)	Legge l' <i>UID</i> effettivo del processo corrente.
gid_t getgid(void)	Legge il <i>GID</i> reale del processo corrente.
gid_t getegid(void)	Legge il <i>GID</i> effettivo del processo corrente.
Le funzioni ritornano i rispettivi identificativi del processo corrente, e non sono previste condizioni di errore.	

In generale l'uso di privilegi superiori, ottenibile con un *UID effettivo* diverso da quello reale, deve essere limitato il più possibile, per evitare abusi e problemi di sicurezza, per questo occorre anche un meccanismo che consenta ad un programma di rilasciare gli eventuali

maggiori privilegi necessari, una volta che si siano effettuate le operazioni per i quali erano richiesti, e a poterli eventualmente recuperare in caso servano di nuovo.

Questo in Linux viene fatto usando altri due gruppi di identificatori, il *saved* ed il *filesystem*. Il primo gruppo è lo stesso usato in SVr4, e previsto dallo standard POSIX quando è definita `_POSIX_SAVED_IDS`,¹⁹ il secondo gruppo è specifico di Linux e viene usato per migliorare la sicurezza con NFS (il *Network File System*, protocollo che consente di accedere ai file via rete).

L'*UID salvato* ed il *GID salvato* sono copie dell'*UID effettivo* e del *GID effettivo* del processo padre, e vengono impostati dalla funzione `exec` all'avvio del processo, come copie dell'*UID effettivo* e del *GID effettivo* dopo che questi sono stati impostati tenendo conto di eventuali permessi *suid* o *sgid* (su cui torneremo in sez. 4.4.2). Essi quindi consentono di tenere traccia di quale fossero utente e gruppo effettivi all'inizio dell'esecuzione di un nuovo programma.

L'*UID di filesystem* e il *GID di filesystem* sono un'estensione introdotta in Linux per rendere più sicuro l'uso di NFS (torneremo sull'argomento in sez. 3.2.2). Essi sono una replica dei corrispondenti identificatori del gruppo *effective*, ai quali si sostituiscono per tutte le operazioni di verifica dei permessi relativi ai file (trattate in sez. 4.4.1). Ogni cambiamento effettuato sugli identificatori effettivi viene automaticamente riportato su di essi, per cui in condizioni normali si può tranquillamente ignorarne l'esistenza, in quanto saranno del tutto equivalenti ai precedenti.

3.2.2 Le funzioni di gestione degli identificatori dei processi

Le funzioni di sistema più comuni che vengono usate per cambiare identità (cioè utente e gruppo di appartenenza) ad un processo, e che come accennato in sez. 3.2.1 seguono la semantica POSIX che prevede l'esistenza dell'*UID salvato* e del *GID salvato*, sono rispettivamente `setuid` e `setgid`; i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int setuid(uid_t uid)
int setgid(gid_t gid)
```

Imposta l'*UID* del processo corrente.

Imposta il *GID* del processo corrente.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` uno dei valori:

EAGAIN (solo per `setuid`) la chiamata cambierebbe l'*UID* reale ma il kernel non dispone temporaneamente delle risorse per farlo, oppure, per i kernel precedenti il 3.1, il cambiamento dell'*UID* reale farebbe superare il limite per il numero dei processi `RLIMIT_NPROC` (vedi sez. 6.3.2).

EINVAL il valore di dell'argomento non è valido per il *namespace* corrente (vedi sez. 9.3.2).

EPERM non si hanno i permessi per l'operazione richiesta.

Il funzionamento di queste due funzioni è analogo, per cui considereremo solo la prima, la seconda si comporta esattamente allo stesso modo facendo riferimento al *GID* invece che all'*UID*. Gli eventuali *GID* supplementari non vengono modificati.

L'effetto della chiamata è diverso a seconda dei privilegi del processo; se l'*UID* effettivo è zero (cioè è quello dell'amministratore di sistema o il processo ha la capacità `CAP_SETUID`) allora tutti gli identificatori (*real*, *effective* e *saved*) vengono impostati al valore specificato

¹⁹in caso si abbia a cuore la portabilità del programma su altri Unix è buona norma controllare sempre la disponibilità di queste funzioni controllando se questa costante è definita.

da *uid*, altrimenti viene impostato solo l'*UID* effettivo, e soltanto se il valore specificato corrisponde o all'*UID* reale o all'*UID* salvato, ottenendo un errore di *EPERM* negli altri casi.

E' importante notare che la funzione può fallire, con *EAGAIN*,²⁰ anche quando viene invocata da un processo con privilegi di amministratore per cambiare il proprio *UID* reale, sia per una temporanea indisponibilità di risorse del kernel, sia perché l'utente di cui si vuole assumere l'*UID* andrebbe a superare un eventuale limite sul numero di processi (il limite *RLIMIT_NPROC*, che tratteremo in sez. 6.3.2), pertanto occorre sempre verificare lo stato di uscita della funzione.

Non controllare questo tipo di errori perché si presume che la funzione abbia sempre successo quando si hanno i privilegi di amministratore può avere conseguente devastanti per la sicurezza, in particolare quando la si usa per cedere i suddetti privilegi ed eseguire un programma per conto di un utente non privilegiato.

E' per diminuire l'impatto di questo tipo di disattenzioni che a partire dal kernel 3.1 il comportamento di *setuid* e di tutte le analoghe funzioni che tratteremo nel seguito di questa sezione è stato modificato e nel caso di superamento del limite sulle risorse esse hanno comunque successo. Quando questo avviene il processo assume comunque il nuovo *UID* ed il controllo sul superamento di *RLIMIT_NPROC* viene posticipato ad una eventuale successiva invocazione di *execve* (essendo questo poi il caso d'uso più comune). In tal caso, se alla chiamata ancora sussiste la situazione di superamento del limite, sarà *execve* a fallire con un errore di *EAGAIN*.²¹

Come accennato l'uso principale di queste funzioni è quello di poter consentire ad un programma con i bit *suid* o *sgid* impostati (vedi sez. 4.4.2) di riportare l'*UID* effettivo a quello dell'utente che ha lanciato il programma, effettuare il lavoro che non necessita di privilegi aggiuntivi, ed eventualmente tornare indietro.

Come esempio per chiarire l'uso di queste funzioni prendiamo quello con cui viene gestito l'accesso al file */var/run/utmp*. In questo file viene registrato chi sta usando il sistema al momento corrente; chiaramente non può essere lasciato aperto in scrittura a qualunque utente, che potrebbe falsificare la registrazione.

Per questo motivo questo file (e l'analogo */var/log/wtmp* su cui vengono registrati login e logout) appartengono ad un gruppo dedicato (in genere *utmp*) ed i programmi che devono accedervi (ad esempio tutti i programmi di terminale in X, o il programma *screen* che crea terminali multipli su una console) appartengono a questo gruppo ed hanno il bit *sgid* impostato.

Quando uno di questi programmi (ad esempio *xterm*) viene lanciato, la situazione degli identificatori è la seguente:

$$\begin{aligned} \text{group-ID reale} &= GID \text{ (del chiamante)} \\ \text{group-ID effettivo} &= utmp \\ \text{group-ID salvato} &= utmp \end{aligned}$$

in questo modo, dato che il *group-ID effettivo* è quello giusto, il programma può accedere a */var/run/utmp* in scrittura ed aggiornarlo. A questo punto il programma può eseguire una *setgid(getgid())* per impostare il *group-ID effettivo* a quello dell'utente (e dato che il *group-ID reale* corrisponde la funzione avrà successo), in questo modo non sarà possibile

²⁰non affronteremo qui l'altro caso di errore, che può avvenire solo quando si esegue la funzione all'interno di un diverso *user namespace*, argomento su cui torneremo in sez. 9.3.2 ma la considerazione di controllare sempre lo stato di uscita si applica allo stesso modo.

²¹che pertanto, a partire dal kernel 3.1, può restituire anche questo errore, non presente in altri sistemi *unix-like*.

lanciare dal terminale programmi che modificano detto file, in tal caso infatti la situazione degli identificatori sarebbe:

<i>group-ID reale</i>	=	<i>GID</i> (invariato)
<i>group-ID effettivo</i>	=	<i>GID</i>
<i>group-ID salvato</i>	=	<i>utmp</i> (invariato)

e ogni processo lanciato dal terminale avrebbe comunque *GID* come *group-ID effettivo*. All'uscita dal terminale, per poter di nuovo aggiornare lo stato di */var/run/utmp* il programma eseguirà una *setgid(utmp)* (dove *utmp* è il valore numerico associato al gruppo *utmp*, ottenuto ad esempio con una precedente *getegid*), dato che in questo caso il valore richiesto corrisponde al *group-ID salvato* la funzione avrà successo e riporterà la situazione a:

<i>group-ID reale</i>	=	<i>GID</i> (invariato)
<i>group-ID effettivo</i>	=	<i>utmp</i>
<i>group-ID salvato</i>	=	<i>utmp</i> (invariato)

consentendo l'accesso a */var/run/utmp*.

Occorre però tenere conto che tutto questo non è possibile con un processo con i privilegi di amministratore, in tal caso infatti l'esecuzione di una *setuid* comporta il cambiamento di tutti gli identificatori associati al processo, rendendo impossibile riguadagnare i privilegi di amministratore. Questo comportamento è corretto per l'uso che ne fa un programma come *login* una volta che crea una nuova shell per l'utente, ma quando si vuole cambiare soltanto l'*UID* effettivo del processo per cedere i privilegi occorre ricorrere ad altre funzioni.

Le due funzioni di sistema *setreuid* e *setregid* derivano da BSD che, non supportando (almeno fino alla versione 4.3+BSD) gli identificatori del gruppo *saved*, le usa per poter scambiare fra di loro *effective* e *real*; i rispettivi prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int setreuid(uid_t ruid, uid_t euid)
    Imposta UID reale e UID effettivo del processo corrente.
int setregid(gid_t rgid, gid_t egid)
    Imposta GID reale e GID effettivo del processo corrente.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso *errno* assume i valori visti per *setuid/setgid*.

Le due funzioni sono identiche, quanto diremo per la prima riguardo gli *UID* si applica alla seconda per i *GID*. La funzione *setreuid* imposta rispettivamente l'*UID* reale e l'*UID* effettivo del processo corrente ai valori specificati da *ruid* e *euid*.

I processi non privilegiati possono impostare solo valori che corrispondano o al loro *UID* effettivo o a quello reale o a quello salvato, valori diversi comportano il fallimento della chiamata. L'amministratore invece può specificare un valore qualunque. Specificando un argomento di valore -1 l'identificatore corrispondente verrà lasciato inalterato.

Con queste funzioni si possono scambiare fra loro gli *UID* reale ed effettivo, e pertanto è possibile implementare un comportamento simile a quello visto in precedenza per *setgid*, cedendo i privilegi con un primo scambio, e recuperandoli, una volta eseguito il lavoro non privilegiato, con un secondo scambio.

In questo caso però occorre porre molta attenzione quando si creano nuovi processi nella fase intermedia in cui si sono scambiati gli identificatori, in questo caso infatti essi avranno un *UID* reale privilegiato, che dovrà essere esplicitamente eliminato prima di porre in esecuzione

un nuovo programma, occorrerà cioè eseguire un'altra chiamata dopo la `fork` e prima della `exec` per uniformare l'*UID* reale a quello effettivo, perché in caso contrario il nuovo programma potrebbe a sua volta effettuare uno scambio e riottenere dei privilegi non previsti.

Lo stesso problema di propagazione dei privilegi ad eventuali processi figli si pone anche per l'*UID* salvato. Ma la funzione `setreuid` deriva da un'implementazione di sistema che non ne prevede la presenza, e quindi non è possibile usarla per correggere la situazione come nel caso precedente. Per questo motivo in Linux tutte le volte che si imposta un qualunque valore diverso da quello dall'*UID* reale corrente, l'*UID* salvato viene automaticamente uniformato al valore dell'*UID* effettivo.

Altre due funzioni di sistema, `seteuid` e `setegid`, sono un'estensione dello standard POSIX.1, ma sono comunque supportate dalla maggior parte degli Unix, esse vengono usate per cambiare gli identificatori del gruppo *effective* ed i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int seteuid(uid_t uid)
int setegid(gid_t gid)
```

Imposta l'*UID* effettivo del processo corrente.
Imposta il *GID* effettivo del processo corrente.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assume i valori visti per `setuid/setgid` tranne `EAGAIN`.

Ancora una volta le due funzioni sono identiche, e quanto diremo per la prima riguardo gli *UID* si applica allo stesso modo alla seconda per i *GID*. Con `seteuid` gli utenti normali possono impostare l'*UID* effettivo solo al valore dell'*UID* reale o dell'*UID* salvato, l'amministratore può specificare qualunque valore. Queste funzioni sono usate per permettere all'amministratore di impostare solo l'*UID* effettivo, dato che l'uso normale di `setuid` comporta l'impostazione di tutti gli identificatori.

Le due funzioni di sistema `setresuid` e `setresgid` sono invece un'estensione introdotta in Linux (a partire dal kernel 2.1.44) e permettono un completo controllo su tutti e tre i gruppi di identificatori (*real*, *effective* e *saved*), i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int setresuid(uid_t ruid, uid_t euid, uid_t suid)
int setresgid(gid_t rgid, gid_t egid, gid_t sgid)
```

Imposta l'*UID* reale, effettivo e salvato del processo corrente.
Imposta il *GID* reale, effettivo e salvato del processo corrente.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assume i valori visti per `setuid/setgid`.

Di nuovo le due funzioni sono identiche e quanto detto per la prima riguardo gli *UID* si applica alla seconda per i *GID*. La funzione `setresuid` imposta l'*UID* reale, l'*UID* effettivo e l'*UID* salvato del processo corrente ai valori specificati rispettivamente dagli argomenti `ruid`, `euid` e `suid`. I processi non privilegiati possono cambiare uno qualunque degli *UID* solo ad un valore corrispondente o all'*UID* reale, o a quello effettivo o a quello salvato, l'amministratore può specificare i valori che vuole. Un valore di -1 per un qualunque argomento lascia inalterato l'identificatore corrispondente.

Per queste funzioni di sistema esistono anche due controparti, `getresuid` e `getresgid`,²² che permettono di leggere in blocco i vari identificatori; i loro prototipi sono:

²²le funzioni non sono standard, anche se appaiono in altri kernel, su Linux sono presenti dal kernel 2.1.44 e con le versioni della *glibc* a partire dalla 2.3.2, definendo la macro `_GNU_SOURCE`.

```
#include <unistd.h>
#include <sys/types.h>
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid)
    Legge l'UID reale, effettivo e salvato del processo corrente.
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid)
    Legge il GID reale, effettivo e salvato del processo corrente.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` può assumere solo il valore `EFAULT` se gli indirizzi delle variabili di ritorno non sono validi.

Anche queste funzioni sono un'estensione specifica di Linux, e non richiedono nessun privilegio. I valori sono restituiti negli argomenti, che vanno specificati come puntatori (è un altro esempio di *value result argument*). Si noti che queste funzioni sono le uniche in grado di leggere gli identificatori del gruppo *saved*.

Infine le funzioni `setfsuid` e `setfsgid` servono per impostare gli identificatori del gruppo *filesystem* che sono usati da Linux per il controllo dell'accesso ai file. Come già accennato in sez. 3.2.1 Linux definisce questo ulteriore gruppo di identificatori, che in circostanze normali sono assolutamente equivalenti a quelli del gruppo *effective*, dato che ogni cambiamento di questi ultimi viene immediatamente riportato su di essi.

C'è un solo caso in cui si ha necessità di introdurre una differenza fra gli identificatori dei gruppi *effective* e *filesystem*, ed è per ovviare ad un problema di sicurezza che si presenta quando si deve implementare un server NFS.

Il server NFS infatti deve poter cambiare l'identificatore con cui accede ai file per assumere l'identità del singolo utente remoto, ma se questo viene fatto cambiando l'*UID* effettivo o l'*UID* reale il server si espone alla ricezione di eventuali segnali ostili da parte dell'utente di cui ha temporaneamente assunto l'identità. Cambiando solo l'*UID* di *filesystem* si ottengono i privilegi necessari per accedere ai file, mantenendo quelli originari per quanto riguarda tutti gli altri controlli di accesso, così che l'utente non possa inviare segnali al server NFS.

Le due funzioni di sistema usate appositamente per cambiare questi identificatori sono `setfsuid` e `setfsgid` ovviamente sono specifiche di Linux e non devono essere usate se si intendono scrivere programmi portabili; i loro prototipi sono:

```
#include <sys/fsuid.h>
int setfsuid(uid_t fsuid)
    Imposta l'UID di filesystem del processo corrente.
int setfsgid(gid_t fsgid)
    Legge il GID di filesystem del processo corrente.
```

Le funzioni restituiscono sia in caso di successo che di errore il valore corrente dell'identificativo, e in caso di errore non viene impostato nessun codice in `errno`.

Le due funzioni sono analoghe ed usano il valore passato come argomento per effettuare l'impostazione dell'identificativo. Le funzioni hanno successo solo se il processo chiamante ha i privilegi di amministratore o, per gli altri utenti, se il valore specificato coincide con uno dei quelli del gruppo *real*, *effective* o *saved*.

Il problema di queste funzioni è che non restituiscono un codice di errore e non c'è modo di sapere (con una singola chiamata) di sapere se hanno avuto successo o meno, per verificarlo occorre eseguire una chiamata aggiuntiva passando come argomento -1 (un valore impossibile per un identificativo), così fallendo si può di ottenere il valore corrente e verificare se è cambiato.

3.2.3 Le funzioni per la gestione dei gruppi associati a un processo

Le ultime funzioni che esamineremo sono quelle che permettono di operare sui gruppi supplementari cui un utente può appartenere. Ogni processo può avere almeno `NGROUPS_MAX` gruppi supplementari²³ in aggiunta al gruppo primario; questi vengono ereditati dal processo padre e possono essere cambiati con queste funzioni.

La funzione di sistema che permette di leggere i gruppi supplementari associati ad un processo è `getgroups`; questa funzione è definita nello standard POSIX.1, ed il suo prototipo è:

```
#include <sys/types.h>
#include <unistd.h>
int getgroups(int size, gid_t list[])
```

Legge gli identificatori dei gruppi supplementari.

La funzione ritorna il numero di gruppi letti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EFAULT` `list` non ha un indirizzo valido.

`EINVAL` il valore di `size` è diverso da zero ma minore del numero di gruppi supplementari del processo.

La funzione legge gli identificatori dei gruppi supplementari del processo sul vettore `list` che deve essere di dimensione pari a `size`. Non è specificato se la funzione inserisca o meno nella lista il `GID` effettivo del processo. Se si specifica un valore di `size` uguale a 0 allora l'argomento `list` non viene modificato, ma si ottiene dal valore di ritorno il numero di gruppi supplementari.

Una seconda funzione, `getgrouplist`, può invece essere usata per ottenere tutti i gruppi a cui appartiene utente identificato per nome; il suo prototipo è:

```
#include <grp.h>
int getgrouplist(const char *user, gid_t group, gid_t *groups, int *ngroups)
```

Legge i gruppi cui appartiene un utente.

La funzione ritorna il numero di gruppi ottenuto in caso di successo e `-1` per un errore, che avviene solo quando il numero di gruppi è maggiore di quelli specificati con `ngroups`.

La funzione esegue una scansione del database dei gruppi (si veda sez. 6.2.1) per leggere i gruppi supplementari dell'utente specificato per nome (e non con un `UID`) nella stringa passata con l'argomento `user`. Ritorna poi nel vettore `groups` la lista dei `GID` dei gruppi a cui l'utente appartiene. Si noti che `ngroups`, che in ingresso deve indicare la dimensione di `group`, è passato come *value result argument* perché, qualora il valore specificato sia troppo piccolo, la funzione ritorna `-1`, passando comunque indietro il numero dei gruppi trovati, in modo da poter ripetere la chiamata con un vettore di dimensioni adeguate.

Infine per impostare i gruppi supplementari di un processo ci sono due funzioni, che possono essere usate solo se si hanno i privilegi di amministratore.²⁴ La prima delle due è la funzione di sistema `setgroups`,²⁵ ed il suo prototipo è:

²³il numero massimo di gruppi secondari può essere ottenuto con `sysconf` (vedi sez. 6.1.1), leggendo il parametro `_SC_NGROUPS_MAX`.

²⁴e più precisamente se si ha la *capability* `CAP_SETGID`.

²⁵la funzione è definita in BSD e SRv4, ma a differenza di `getgroups` non è stata inclusa in POSIX.1-2001, per poterla utilizzare deve essere definita la macro `_BSD_SOURCE`.

<pre>#include <grp.h> int setgroups(size_t size, gid_t *list)</pre>		Imposta i gruppi supplementari del processo.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:		
EFAULT	list non ha un indirizzo valido.	
EINVAL	il valore di <code>size</code> è maggiore del valore massimo consentito di gruppi supplementari.	
EPERM	il processo non ha i privilegi di amministratore.	

La funzione imposta i gruppi supplementari del processo corrente ai valori specificati nel vettore passato con l'argomento `list`, di dimensioni date dall'argomento `size`. Il numero massimo di gruppi supplementari che si possono impostare è un parametro di sistema, che può essere ricavato con le modalità spiegate in sez. 6.1.

Se invece si vogliono impostare i gruppi supplementari del processo a quelli di un utente specifico, si può usare la funzione `initgroups` il cui prototipo è:

<pre>#include <sys/types.h> #include <grp.h> int initgroups(const char *user, gid_t group)</pre>		Inizializza la lista dei gruppi supplementari.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:		
ENOMEM	non c'è memoria sufficiente per allocare lo spazio per informazioni dei gruppi.	
EPERM	il processo non ha i privilegi di amministratore.	

La funzione esegue la scansione del database dei gruppi (usualmente `/etc/group`) cercando i gruppi di cui è membro l'utente `user` (di nuovo specificato per nome e non per `UID`) con cui costruisce una lista di gruppi supplementari, a cui aggiunge anche `group`, infine imposta questa lista per il processo corrente usando `setgroups`. Si tenga presente che sia `setgroups` che `initgroups` non sono definite nello standard POSIX.1 e che pertanto non è possibile utilizzarle quando si definisce `_POSIX_SOURCE` o si compila con il flag `-ansi`, è pertanto meglio evitarle se si vuole scrivere codice portabile.

3.3 La gestione della priorità dei processi

In questa sezione tratteremo più approfonditamente i meccanismi con il quale lo *scheduler* assegna la CPU ai vari processi attivi. In particolare prenderemo in esame i vari meccanismi con cui viene gestita l'assegnazione del tempo di CPU, ed illustreremo le varie funzioni di gestione. Tratteremo infine anche le altre priorità dei processi (come quelle per l'accesso a disco) divenute disponibili con i kernel più recenti.

3.3.1 I meccanismi di *scheduling*

La scelta di un meccanismo che sia in grado di distribuire in maniera efficace il tempo di CPU per l'esecuzione dei processi è sempre una questione delicata, ed oggetto di numerose ricerche; in generale essa dipende in maniera essenziale anche dal tipo di utilizzo che deve essere fatto del sistema, per cui non esiste un meccanismo che sia valido per tutti gli usi.

La caratteristica specifica di un sistema *multitasking* come Linux è quella del cosiddetto *preemptive multitasking*: questo significa che al contrario di altri sistemi (che usano invece il cosiddetto *cooperative multitasking*) non sono i singoli processi, ma il kernel stesso a decidere quando la CPU deve essere passata ad un altro processo. Come accennato in sez. 3.1.1 questa

scelta viene eseguita da una sezione apposita del kernel, lo *scheduler*, il cui scopo è quello di distribuire al meglio il tempo di CPU fra i vari processi.

La cosa è resa ancora più complicata dal fatto che con le architetture multi-processore si deve anche scegliere quale sia la CPU più opportuna da utilizzare.²⁶ Tutto questo comunque appartiene alle sottigliezze dell'implementazione del kernel; dal punto di vista dei programmi che girano in *user space*, anche quando si hanno più processori (e dei processi che sono eseguiti davvero in contemporanea), le politiche di *scheduling* riguardano semplicemente l'allocazione della risorsa *tempo di esecuzione*, la cui assegnazione sarà governata dai meccanismi di scelta delle priorità che restano gli stessi indipendentemente dal numero di processori.

Si tenga conto poi che i processi non devono solo eseguire del codice: ad esempio molto spesso saranno impegnati in operazioni di I/O, o potranno venire bloccati da un comando dal terminale, o sospesi per un certo periodo di tempo. In tutti questi casi la CPU diventa disponibile ed è compito dello kernel provvedere a mettere in esecuzione un altro processo.

Tutte queste possibilità sono caratterizzate da un diverso *stato* del processo; in Linux un processo può trovarsi in uno degli stati riportati in tab. 3.8; ma soltanto i processi che sono nello stato *runnable* concorrono per l'esecuzione. Questo vuol dire che, qualunque sia la sua priorità, un processo non potrà mai essere messo in esecuzione fintanto che esso si trova in uno qualunque degli altri stati.

Stato	STAT	Descrizione
<i>runnable</i>	R	Il processo è in esecuzione o è pronto ad essere eseguito (in attesa che gli venga assegnata la CPU).
<i>sleep</i>	S	Il processo è in attesa di un risposta dal sistema, ma può essere interrotto da un segnale.
<i>uninterruptible sleep</i>	D	Il processo è in attesa di un risposta dal sistema (in genere per I/O), e non può essere interrotto in nessuna circostanza.
<i>stopped</i>	T	Il processo è stato fermato con un SIGSTOP, o è tracciato.
<i>zombie</i>	Z	Il processo è terminato ma il suo stato di terminazione non è ancora stato letto dal padre.
<i>killable</i>	D	Un nuovo stato introdotto con il kernel 2.6.25, sostanzialmente identico all' <i>uninterruptible sleep</i> con la sola differenza che il processo può terminato con SIGKILL (usato per lo più per NFS).

Tabella 3.8: Elenco dei possibili stati di un processo in Linux, nella colonna STAT si è riportata la corrispondente lettera usata dal comando ps nell'omonimo campo.

Si deve quindi tenere presente che l'utilizzo della CPU è soltanto una delle risorse che sono necessarie per l'esecuzione di un programma, e a seconda dello scopo del programma non è detto neanche che sia la più importante, dato che molti programmi dipendono in maniera molto più critica dall'I/O. Per questo motivo non è affatto detto che dare ad un programma la massima priorità di esecuzione abbia risultati significativi in termini di prestazioni.

Il meccanismo tradizionale di *scheduling* di Unix (che tratteremo in sez. 3.3.2) è sempre stato basato su delle *priorità dinamiche*, in modo da assicurare che tutti i processi, anche i meno importanti, potessero ricevere un po' di tempo di CPU. In sostanza quando un processo ottiene la CPU la sua priorità viene diminuita. In questo modo alla fine, anche un processo con priorità iniziale molto bassa, finisce per avere una priorità sufficiente per essere eseguito.

Lo standard POSIX.1b però ha introdotto il concetto di *priorità assoluta*, (chiamata anche *priorità statica*, in contrapposizione alla normale priorità dinamica), per tenere conto dei

²⁶nei processori moderni la presenza di ampie cache può rendere poco efficiente trasferire l'esecuzione di un processo da una CPU ad un'altra, per cui effettuare la migliore scelta fra le diverse CPU non è banale.

sistemi *real-time*,²⁷ in cui è vitale che i processi che devono essere eseguiti in un determinato momento non debbano aspettare la conclusione di altri che non hanno questa necessità.

Il concetto di priorità assoluta dice che quando due processi si contendono l'esecuzione, vince sempre quello con la priorità assoluta più alta. Ovviamente questo avviene solo per i processi che sono pronti per essere eseguiti (cioè nello stato *runnable*). La priorità assoluta viene in genere indicata con un numero intero, ed un valore più alto comporta una priorità maggiore. Su questa politica di *scheduling* torneremo in sez. 3.3.3.

In generale quello che succede in tutti gli Unix moderni è che ai processi normali viene sempre data una priorità assoluta pari a zero, e la decisione di assegnazione della CPU è fatta solo con il meccanismo tradizionale della priorità dinamica. In Linux tuttavia è possibile assegnare anche una priorità assoluta, nel qual caso un processo avrà la precedenza su tutti gli altri di priorità inferiore, che saranno eseguiti solo quando quest'ultimo non avrà bisogno della CPU.

3.3.2 Il meccanismo di *scheduling* standard

A meno che non si abbiano esigenze specifiche,²⁸ l'unico meccanismo di *scheduling* con il quale si avrà a che fare è quello tradizionale, che prevede solo priorità dinamiche. È di questo che, di norma, ci si dovrà preoccupare nella programmazione. Come accennato in Linux i processi ordinari hanno tutti una priorità assoluta nulla; quello che determina quale, fra tutti i processi in attesa di esecuzione, sarà eseguito per primo, è la cosiddetta *priorità dinamica*, quella che viene mostrata nella colonna PR del comando `top`, che è chiamata così proprio perché varia nel corso dell'esecuzione di un processo.

Il meccanismo usato da Linux è in realtà piuttosto complesso,²⁹ ma a grandi linee si può dire che ad ogni processo è assegnata una *time-slice*, cioè un intervallo di tempo (letteralmente una fetta) per il quale, a meno di eventi esterni, esso viene eseguito senza essere interrotto. Inoltre la priorità dinamica viene calcolata dallo *scheduler* a partire da un valore iniziale che viene *diminuito* tutte le volte che un processo è in stato *runnable* ma non viene posto in esecuzione.³⁰

Lo *scheduler* infatti mette sempre in esecuzione, fra tutti i processi in stato *runnable*, quello che ha il valore di priorità dinamica più basso; con le priorità dinamiche il significato del valore numerico ad esse associato è infatti invertito, un valore più basso significa una priorità maggiore. Il fatto che questo valore venga diminuito quando un processo non viene posto in esecuzione pur essendo pronto, significa che la priorità dei processi che non ottengono l'uso del processore viene progressivamente incrementata, così che anche questi alla fine hanno la possibilità di essere eseguiti.

²⁷per sistema *real-time* si intende un sistema in grado di eseguire operazioni in un tempo ben determinato; in genere si tende a distinguere fra l'*hard real-time* in cui è necessario che i tempi di esecuzione di un programma siano determinabili con certezza assoluta (come nel caso di meccanismi di controllo di macchine, dove uno sfioramento dei tempi avrebbe conseguenze disastrose), e *soft-real-time* in cui un occasionale sfioramento è ritenuto accettabile.

²⁸per alcune delle quali sono state introdotte delle varianti specifiche.

²⁹e dipende strettamente dalla versione di kernel; in particolare a partire dalla serie 2.6.x lo *scheduler* è stato riscritto completamente, con molte modifiche susseguites per migliorarne le prestazioni, per un certo periodo ed è stata anche introdotta la possibilità di usare diversi algoritmi, selezionabili sia in fase di compilazione, che, nelle versioni più recenti, all'avvio (addirittura è stato ideato un sistema modulare che permette di cambiare lo *scheduler* a sistema attivo).

³⁰in realtà il calcolo della priorità dinamica e la conseguente scelta di quale processo mettere in esecuzione avviene con un algoritmo molto più complicato, che tiene conto anche della *interattività* del processo, utilizzando diversi fattori, questa è una brutale semplificazione per rendere l'idea del funzionamento, per una trattazione più dettagliata dei meccanismi di funzionamento dello *scheduler*, anche se non aggiornatissima, si legga il quarto capitolo di [?].

Sia la dimensione della *time-slice* che il valore di partenza della priorità dinamica sono determinate dalla cosiddetta *nice* (o *nice*ness) del processo.³¹ L'origine del nome di questo parametro sta nel fatto che generalmente questo viene usato per *diminuire* la priorità di un processo, come misura di cortesia nei confronti degli altri. I processi infatti vengono creati dal sistema con un valore nullo e nessuno è privilegiato rispetto agli altri. Specificando un valore di *nice* positivo si avrà una *time-slice* più breve ed un valore di priorità dinamica iniziale più alto, mentre un valore negativo darà una *time-slice* più lunga ed un valore di priorità dinamica iniziale più basso.

Esistono diverse funzioni che consentono di indicare un valore di *nice* di un processo; la più semplice è *nice*, che opera sul processo corrente, il suo prototipo è:

<pre>#include <unistd.h> int nice(int inc)</pre>	Aumenta il valore di <i>nice</i> del processo corrente.
La funzione ritorna il nuovo valore di <i>nice</i> in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà uno dei valori: EPERM non si ha il permesso di specificare un valore di <i>inc</i> negativo.	

L'argomento *inc* indica l'incremento da effettuare rispetto al valore di *nice* corrente, che può assumere valori compresi fra *PRIO_MIN* e *PRIO_MAX*; nel caso di Linux sono fra -20 e 19 ,³² ma per *inc* si può specificare un valore qualunque, positivo o negativo, ed il sistema provvederà a troncare il risultato nell'intervallo consentito. Valori positivi comportano maggiore *cortesia* e cioè una diminuzione della priorità, valori negativi comportano invece un aumento della priorità. Con i kernel precedenti il 2.6.12 solo l'amministratore³³ può specificare valori negativi di *inc* che permettono di aumentare la priorità di un processo, a partire da questa versione è consentito anche agli utenti normali alzare (entro certi limiti, che vedremo in sez. 6.3.2) la priorità dei propri processi.

Gli standard SUSv2 e POSIX.1 prevedono che la funzione ritorni il nuovo valore di *nice* del processo; tuttavia la *system call* di Linux non segue questa convenzione e restituisce sempre 0 in caso di successo e -1 in caso di errore; questo perché -1 è anche un valore di *nice* legittimo e questo comporta una confusione con una eventuale condizione di errore. La *system call* originaria inoltre non consente, se non dotati di adeguati privilegi, di diminuire un valore di *nice* precedentemente innalzato.

Fino alla *glibc* 2.2.4 la funzione di libreria riportava direttamente il risultato dalla *system call*, violando lo standard, per cui per ottenere il nuovo valore occorreva una successiva chiamata alla funzione *getpriority*. A partire dalla *glibc* 2.2.4 *nice* è stata reimplementata e non viene più chiamata la omonima *system call*, con questa versione viene restituito come valore di ritorno il valore di *nice*, come richiesto dallo standard.³⁴ In questo caso l'unico modo per rilevare in maniera affidabile una condizione di errore è quello di azzerare *errno* prima della chiamata della funzione e verificarne il valore quando *nice* restituisce -1 .

Per leggere il valore di *nice* di un processo occorre usare la funzione di sistema *getpriority*, derivata da BSD; il suo prototipo è:

³¹questa è una delle tante proprietà che ciascun processo si porta dietro, essa viene ereditata dai processi figli e mantenuta attraverso una *exec*; fino alla serie 2.4 essa era mantenuta nell'omonimo campo *nice* della *task_struct*, con la riscrittura dello *scheduler* eseguita nel 2.6 viene mantenuta nel campo *static_prio* come per le priorità statiche.

³²in realtà l'intervallo varia a seconda delle versioni di kernel, ed è questo a partire dal kernel 1.3.43, anche se oggi si può avere anche l'intervallo fra -20 e 20 .

³³o un processo con la *capability* *CAP_SYS_NICE*, vedi sez. 9.1.1.

³⁴questo viene fatto chiamando al suo interno *setpriority*, che tratteremo a breve.

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who)
```

Legge un valore di *nice*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EINVAL il valore di *which* non è uno di quelli elencati in tab. 3.9.

ESRCH non c'è nessun processo che corrisponda ai valori di *which* e *who*.

La funzione permette, a seconda di quanto specificato nell'argomento *which*, di leggere il valore di *nice* o di un processo, o di un gruppo di processi (vedi sez. 8.1.2) o di un utente, indicati con l'argomento *who*. Nelle vecchie versioni può essere necessario includere anche *sys/time.h*, questo non è più necessario con versioni recenti delle librerie, ma è comunque utile per portabilità.

I valori possibili per *which*, ed il tipo di valore che occorre usare in corrispondenza per *who*, solo elencati nella legenda di tab. 3.9 insieme ai relativi significati. Usare un valore nullo per *who* indica, a seconda della corrispondente indicazione usata per *which*, il processo, il gruppo di processi o l'utente correnti.

which	who	Significato
PRIO_PROCESS	pid_t	processo
PRIO_PRGR	pid_t	<i>process group</i> (vedi sez. 8.1.2)
PRIO_USER	uid_t	utente

Tabella 3.9: Legenda del valore dell'argomento *which* e del tipo dell'argomento *who* delle funzioni *getpriority* e *setpriority* per le tre possibili scelte.

In caso di una indicazione che faccia riferimento a più processi, la funzione restituisce la priorità più alta (cioè il valore più basso) fra quelle dei processi corrispondenti. Come per *nice*, -1 è un possibile valore corretto, per cui di nuovo per poter rilevare una condizione di errore è necessario cancellare sempre *errno* prima della chiamata alla funzione e quando si ottiene un valore di ritorno uguale a -1 per verificare che essa resti uguale a zero.

Analoga a *getpriority* è la funzione di sistema *setpriority* che permette di impostare la priorità di uno o più processi; il suo prototipo è:

```
#include <sys/time.h>
#include <sys/resource.h>
int setpriority(int which, int who, int prio)
```

Imposta un valore di *nice*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EACCES si è richiesto un aumento di priorità senza avere sufficienti privilegi.

EINVAL il valore di *which* non è uno di quelli elencati in tab. 3.9.

EPERM un processo senza i privilegi di amministratore ha cercato di modificare la priorità di un processo di un altro utente.

ESRCH non c'è nessun processo che corrisponda ai valori di *which* e *who*.

La funzione imposta la priorità dinamica al valore specificato da *prio* per tutti i processi indicati dagli argomenti *which* e *who*, per i quali valgono le stesse considerazioni fatte per *getpriority* e lo specchio di tab. 3.9.

In questo caso come valore di *prio* deve essere specificato il valore di *nice* da assegnare nell'intervallo fra *PRIO_MIN* (-20) e *PRIO_MAX* (19), e non un incremento (positivo o negativo) come nel caso di *nice*. La funzione restituisce il valore di *nice* assegnato in caso di successo

e -1 in caso di errore, e come per *nice* anche in questo caso per rilevare un errore occorre sempre porre a zero *errno* prima della chiamata della funzione, essendo -1 un valore di *nice* valido.

Si tenga presente che solo l'amministratore³⁵ ha la possibilità di modificare arbitrariamente le priorità di qualunque processo. Un utente normale infatti può modificare solo la priorità dei suoi processi ed in genere soltanto diminuirla. Fino alla versione di kernel 2.6.12 Linux ha seguito le specifiche dello standard SUSv3, e come per tutti i sistemi derivati da SysV veniva richiesto che l'*UID* reale o quello effettivo del processo chiamante corrispondessero all'*UID* reale (e solo a quello) del processo di cui si intendeva cambiare la priorità. A partire dalla versione 2.6.12 è stata adottata la semantica in uso presso i sistemi derivati da BSD (SunOS, Ultrix, *BSD), in cui la corrispondenza può essere anche con l'*UID* effettivo.

Sempre a partire dal kernel 2.6.12 è divenuto possibile anche per gli utenti ordinari poter aumentare la priorità dei propri processi specificando un valore di *prio* negativo. Questa operazione non è possibile però in maniera indiscriminata, ed in particolare può essere effettuata solo nell'intervallo consentito dal valore del limite *RLIMIT_NICE* (torneremo su questo in sez. 6.3.2).

Infine nonostante i valori siano sempre rimasti gli stessi, il significato del valore di *nice* è cambiato parecchio nelle progressive riscritture dello *scheduler* di Linux, ed in particolare a partire dal kernel 2.6.23 l'uso di diversi valori di *nice* ha un impatto molto più forte nella distribuzione della CPU ai processi. Infatti se viene comunque calcolata una priorità dinamica per i processi che non ricevono la CPU, così che anche essi possano essere messi in esecuzione, un alto valore di *nice* corrisponde comunque ad una *time-slice* molto piccola che non cresce comunque, per cui un processo a bassa priorità avrà davvero scarse possibilità di essere eseguito in presenza di processi attivi a priorità più alta.

3.3.3 Il meccanismo di *scheduling real-time*

Come spiegato in sez. 3.3.1 lo standard POSIX.1b ha introdotto le priorità assolute per permettere la gestione di processi *real-time*. In realtà nel caso di Linux non si tratta di un vero *hard real-time*, in quanto in presenza di eventuali interrupt il kernel interrompe l'esecuzione di un processo, qualsiasi sia la sua priorità,³⁶ mentre con l'incorrere in un *page fault* si possono avere ritardi non previsti. Se l'ultimo problema può essere aggirato attraverso l'uso delle funzioni di controllo della memoria virtuale (vedi sez. 2.2.4), il primo non è superabile e può comportare ritardi non prevedibili riguardo ai tempi di esecuzione di qualunque processo.

Nonostante questo, ed in particolare con una serie di miglioramenti che sono stati introdotti nello sviluppo del kernel,³⁷ si può arrivare ad una ottima approssimazione di sistema *real-time* usando le priorità assolute. Occorre farlo però con molta attenzione: se si dà ad un processo una priorità assoluta e questo finisce in un loop infinito, nessun altro processo potrà essere eseguito, ed esso sarà mantenuto in esecuzione permanentemente assorbendo tutta la CPU e senza nessuna possibilità di riottenere l'accesso al sistema. Per questo motivo è sempre opportuno, quando si lavora con processi che usano priorità assolute, tenere attiva una shell

³⁵o più precisamente un processo con la *capability* *CAP_SYS_NICE*, vedi sez. 9.1.1.

³⁶questo a meno che non si siano installate le patch di RTLinux, RTAI o Adeos, con i quali è possibile ottenere un sistema effettivamente *hard real-time*. In tal caso infatti gli interrupt vengono intercettati dall'interfaccia *real-time* (o nel caso di Adeos gestiti dalle code del nano-kernel), in modo da poterli controllare direttamente qualora ci sia la necessità di avere un processo con priorità più elevata di un *interrupt handler*.

³⁷in particolare a partire dalla versione 2.6.18 sono stati inserite nel kernel una serie di modifiche che consentono di avvicinarsi sempre di più ad un vero e proprio sistema *real-time* estendendo il concetto di *preemption* alle operazioni dello stesso kernel; esistono vari livelli a cui questo può essere fatto, ottenibili attivando in fase di compilazione una fra le opzioni *CONFIG_PREEMPT_NONE*, *CONFIG_PREEMPT_VOLUNTARY* e *CONFIG_PREEMPT_DESKTOP*.

cui si sia assegnata la massima priorità assoluta, in modo da poter essere comunque in grado di rientrare nel sistema.

Quando c'è un processo con priorità assoluta lo *scheduler* lo metterà in esecuzione prima di ogni processo normale. In caso di più processi sarà eseguito per primo quello con priorità assoluta più alta. Quando ci sono più processi con la stessa priorità assoluta questi vengono tenuti in una coda e tocca al kernel decidere quale deve essere eseguito. Il meccanismo con cui vengono gestiti questi processi dipende dalla politica di *scheduling* che si è scelta; lo standard ne prevede due:

First In First Out (FIFO)

Il processo viene eseguito fintanto che non cede volontariamente la CPU (con la funzione `sched_yield`), si blocca, finisce o viene interrotto da un processo a priorità più alta. Se il processo viene interrotto da uno a priorità più alta esso resterà in cima alla lista e sarà il primo ad essere eseguito quando i processi a priorità più alta diverranno inattivi. Se invece lo si blocca volontariamente sarà posto in coda alla lista (ed altri processi con la stessa priorità potranno essere eseguiti).

Round Robin (RR)

Il comportamento è del tutto analogo a quello precedente, con la sola differenza che ciascun processo viene eseguito al massimo per un certo periodo di tempo (la cosiddetta *time-slice*) dopo di che viene automaticamente posto in fondo alla coda dei processi con la stessa priorità. In questo modo si ha comunque una esecuzione a turno di tutti i processi, da cui il nome della politica. Solo i processi con la stessa priorità ed in stato *runnable* entrano nel *girotondo*.

Lo standard POSIX.1-2001 prevede una funzione che consenta sia di modificare le politiche di *scheduling*, passando da *real-time* a ordinarie o viceversa, che di specificare, in caso di politiche *real-time*, la eventuale priorità statica; la funzione di sistema è `sched_setscheduler` ed il suo prototipo è:

```
#include <sched.h>
int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p)
    Imposta priorità e politica di scheduling.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL	il valore di <code>policy</code> non esiste o il valore di <code>p</code> non è valido per la politica scelta.
EPERM	il processo non ha i privilegi per attivare la politica richiesta.
ESRCH	il processo <code>pid</code> non esiste.

La funzione esegue l'impostazione per il processo specificato dall'argomento `pid`; un valore nullo di questo argomento esegue l'impostazione per il processo corrente. La politica di *scheduling* è specificata dall'argomento `policy` i cui possibili valori sono riportati in tab. 3.10; la parte alta della tabella indica le politiche *real-time*, quella bassa le politiche ordinarie. Un valore negativo per `policy` mantiene la politica di *scheduling* corrente.

Con le versioni più recenti del kernel sono state introdotte anche delle varianti sulla politica di *scheduling* tradizionale per alcuni carichi di lavoro specifici, queste due nuove politiche sono specifiche di Linux e non devono essere usate se si vogliono scrivere programmi portabili.

La politica `SCHED_BATCH` è una variante della politica ordinaria con la sola differenza che i processi ad essa soggetti non ottengono, nel calcolo delle priorità dinamiche fatto dallo *scheduler*, il cosiddetto bonus di interattività che mira a favorire i processi che si svegliano

Politica	Significato
SCHED_FIFO SCHED_RR	<i>Scheduling real-time</i> con politica <i>FIFO</i> . <i>Scheduling real-time</i> con politica <i>Round Robin</i> .
SCHED_OTHER SCHED_BATCH	<i>Scheduling</i> ordinario. <i>Scheduling</i> ordinario con l'assunzione ulteriore di lavoro <i>CPU intensive</i> (dal kernel 2.6.16).
SCHED_IDLE	<i>Scheduling</i> di priorità estremamente bassa (dal kernel 2.6.23).

Tabella 3.10: Valori dell'argomento policy per la funzione sched_setscheduler.

dallo stato di *sleep*.³⁸ La si usa pertanto, come indica il nome, per processi che usano molta CPU (come programmi di calcolo) che in questo modo, pur non perdendo il loro valore di *nice*, sono leggermente sfavoriti rispetto ai processi interattivi che devono rispondere a dei dati in ingresso.

La politica SCHED_IDLE invece è una politica dedicata ai processi che si desidera siano eseguiti con la più bassa priorità possibile, ancora più bassa di un processo con il minimo valore di *nice*. In sostanza la si può utilizzare per processi che devono essere eseguiti se non c'è niente altro da fare. Va comunque sottolineato che anche un processo SCHED_IDLE avrà comunque una sua possibilità di utilizzo della CPU, sia pure in percentuale molto bassa.

Qualora si sia richiesta una politica *real-time* il valore della priorità statica viene impostato attraverso la struttura sched_param, riportata in fig. 3.5, il cui solo campo attualmente definito è sched_priority. Il campo deve contenere il valore della priorità statica da assegnare al processo; lo standard prevede che questo debba essere assegnato all'interno di un intervallo fra un massimo ed un minimo che nel caso di Linux sono rispettivamente 1 e 99.

```

struct sched_param {
    int sched_priority;
};

```

Figura 3.5: La struttura sched_param.

I processi con politica di *scheduling* ordinaria devono sempre specificare un valore nullo di sched_priority altrimenti si avrà un errore EINVAL, questo valore infatti non ha niente a che vedere con la priorità dinamica determinata dal valore di *nice*, che deve essere impostato con le funzioni viste in precedenza.

Lo standard POSIX.1b prevede che l'intervallo dei valori delle priorità statiche possa essere ottenuto con le funzioni di sistema sched_get_priority_max e sched_get_priority_min, i cui prototipi sono:

```

#include <sched.h>
int sched_get_priority_max(int policy)
int sched_get_priority_min(int policy)

```

Legge il valore massimo di una priorità statica.

Legge il valore minimo di una priorità statica.

Le funzioni ritornano il valore della priorità in caso di successo e -1 per un errore, nel qual caso errno assumerà il valore:

EINVAL il valore di policy non è valido.

³⁸cosa che accade con grande frequenza per i processi interattivi, dato che essi sono per la maggior parte del tempo in attesa di dati in ingresso da parte dell'utente.

Le funzioni ritornano rispettivamente il valore massimo e minimo usabile per la priorità statica di una delle politiche di *scheduling real-time* indicata dall'argomento `policy`.

Si tenga presente che quando si imposta una politica di *scheduling real-time* per un processo o se ne cambia la priorità statica questo viene messo in cima alla lista dei processi con la stessa priorità; questo comporta che verrà eseguito subito, interrompendo eventuali altri processi con la stessa priorità in quel momento in esecuzione.

Il kernel mantiene i processi con la stessa priorità assoluta in una lista, ed esegue sempre il primo della lista, mentre un nuovo processo che torna in stato *runnable* viene sempre inserito in coda alla lista. Se la politica scelta è `SCHED_FIFO` quando il processo viene eseguito viene automaticamente rimesso in coda alla lista, e la sua esecuzione continua fintanto che non viene bloccato da una richiesta di I/O, o non rilascia volontariamente la CPU (in tal caso, tornando nello stato *runnable* sarà in coda alla lista); l'esecuzione viene ripresa subito solo nel caso che esso sia stato interrotto da un processo a priorità più alta.

Solo un processo con i privilegi di amministratore³⁹ può impostare senza restrizioni priorità assolute diverse da zero o politiche `SCHED_FIFO` e `SCHED_RR`. Un utente normale può modificare solo le priorità di processi che gli appartengono; è cioè richiesto che l'*UID* effettivo del processo chiamante corrisponda all'*UID* reale o effettivo del processo indicato con `pid`.

Fino al kernel 2.6.12 gli utenti normali non potevano impostare politiche *real-time* o modificare la eventuale priorità statica di un loro processo. A partire da questa versione è divenuto possibile anche per gli utenti normali usare politiche *real-time* fintanto che la priorità assoluta che si vuole impostare è inferiore al limite `RLIMIT_RTPRIO` (vedi sez. 6.3.2) ad essi assegnato.

Unica eccezione a questa possibilità sono i processi `SCHED_IDLE`, che non possono cambiare politica di *scheduling* indipendentemente dal valore di `RLIMIT_RTPRIO`. Inoltre, in caso di processo già sottoposto ad una politica *real-time*, un utente può sempre, indipendentemente dal valore di `RLIMIT_RTPRIO`, diminuirne la priorità o portarlo ad una politica ordinaria.

Se si intende operare solo sulla priorità statica di un processo si possono usare le due funzioni di sistema `sched_setparam` e `sched_getparam` che consentono rispettivamente di impostarne e leggerne il valore, i loro prototipi sono:

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *param)
                                                    Imposta la priorità statica di un processo.
int sched_getparam(pid_t pid, struct sched_param *param)
                                                    Legge la priorità statica di un processo.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` il valore di `param` non ha senso per la politica usata dal processo.
`EPERM` non si hanno privilegi sufficienti per eseguire l'operazione.
`ESRCH` il processo `pid` non esiste.

Le funzioni richiedono di indicare nell'argomento `pid` il processo su cui operare e usano l'argomento `param` per mantenere il valore della priorità dinamica. Questo è ancora una struttura `sched_param` ed assume gli stessi valori già visti per `sched_setscheduler`.

L'uso di `sched_setparam`, compresi i controlli di accesso che vi si applicano, è del tutto equivalente a quello di `sched_setscheduler` con argomento `policy` uguale a -1. Come per `sched_setscheduler` specificando 0 come valore dell'argomento `pid` si opera sul processo corrente. Benché la funzione sia utilizzabile anche con processi sottoposti a politica ordinaria essa ha senso soltanto per quelli *real-time*, dato che per i primi la priorità statica può essere

³⁹più precisamente con la capacità `CAP_SYS_NICE`, vedi sez. 9.1.1.

soltanto nulla. La disponibilità di entrambe le funzioni può essere verificata controllando la macro `_POSIX_PRIORITY_SCHEDULING` che è definita nell'*header file* `sched.h`.

Se invece si vuole sapere quale è politica di *scheduling* di un processo si può usare la funzione di sistema `sched_getscheduler`, il cui prototipo è:

<pre>#include <sched.h> int sched_getscheduler(pid_t pid)</pre>	Legge la politica di <i>scheduling</i> .
La funzione ritorna la politica di <i>scheduling</i> in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori: EPERM non si hanno privilegi sufficienti per eseguire l'operazione. ESRCH il processo <code>pid</code> non esiste.	

La funzione restituisce il valore, secondo quanto elencato in tab. 3.10, della politica di *scheduling* per il processo specificato dall'argomento `pid`, se questo è nullo viene restituito il valore relativo al processo chiamante.

L'ultima funzione di sistema che permette di leggere le informazioni relative ai processi real-time è `sched_rr_get_interval`, che permette di ottenere la lunghezza della *time-slice* usata dalla politica *round robin*; il suo prototipo è:

<pre>#include <sched.h> int sched_rr_get_interval(pid_t pid, struct timespec *tp)</pre>	Legge la durata della <i>time-slice</i> per lo <i>scheduling round robin</i> .
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori: EINVAL l'argomento <code>pid</code> non è valido. ENOSYS la <i>system call</i> non è presente (solo per kernel arcaici). ESRCH il processo <code>pid</code> non esiste. ed inoltre anche EFAULT nel suo significato generico.	

La funzione restituisce nell'argomento `tp` come una struttura `timespec`, (la cui definizione si può trovare in fig. 4.15) il valore dell'intervallo di tempo usato per la politica *round robin* dal processo indicato da *PID*. Il valore dipende dalla versione del kernel, a lungo infatti questo intervallo di tempo era prefissato e non modificabile ad un valore di 150 millisecondi, restituito indipendentemente dal *PID* indicato.

Con kernel recenti però è possibile ottenere una variazione della *time-slice*, modificando il valore di *nice* del processo (anche se questo non incide assolutamente sulla priorità statica) che come accennato in precedenza modifica il valore assegnato alla *time-slice* di un processo ordinario, che però viene usato anche dai processi *real-time*.

Come accennato ogni processo può rilasciare volontariamente la CPU in modo da consentire agli altri processi di essere eseguiti; la funzione di sistema che consente di fare tutto questo è `sched_yield`, il cui prototipo è:

<pre>#include <sched.h> int sched_yield(void)</pre>	Rilascia volontariamente l'esecuzione.
La funzione ritorna 0 in caso di successo e teoricamente <code>-1</code> per un errore, ma su Linux ha sempre successo.	

Questa funzione ha un utilizzo effettivo soltanto quando si usa lo *scheduling real-time*, e serve a far sì che il processo corrente rilasci la CPU, in modo da essere rimesso in coda alla lista dei processi con la stessa priorità per permettere ad un altro di essere eseguito; se però il processo è l'unico ad essere presente sulla coda l'esecuzione non sarà interrotta. In genere

usano questa funzione i processi con politica `SCHED_FIFO`, per permettere l'esecuzione degli altri processi con pari priorità quando la sezione più urgente è finita.

La funzione può essere utilizzata anche con processi che usano lo *scheduling* ordinario, ma in questo caso il comportamento non è ben definito, e dipende dall'implementazione. Fino al kernel 2.6.23 questo comportava che i processi venissero messi in fondo alla coda di quelli attivi, con la possibilità di essere rimessi in esecuzione entro breve tempo, con l'introduzione del *Completely Fair Scheduler* questo comportamento è cambiato ed un processo che chiama la funzione viene inserito nella lista dei processi inattivi, con un tempo molto maggiore.⁴⁰

L'uso delle funzione nella programmazione ordinaria può essere utile e migliorare le prestazioni generali del sistema quando si è appena rilasciata una risorsa contesa con altri processi, e si vuole dare agli altri una possibilità di approfittarne mettendoli in esecuzione, ma chiamarla senza necessità, specie se questo avviene ripetutamente all'interno di un qualche ciclo, può avere invece un forte impatto negativo per la generazione di *context switch* inutili.

3.3.4 Il controllo dello *scheduler* per i sistemi multiprocessore

Con il supporto dei sistemi multiprocessore sono state introdotte delle funzioni che permettono di controllare in maniera più dettagliata la scelta di quale processore utilizzare per eseguire un certo programma. Uno dei problemi che si pongono nei sistemi multiprocessore è infatti quello del cosiddetto *effetto ping-pong*. Può accadere cioè che lo *scheduler*, quando riavvia un processo precedentemente interrotto scegliendo il primo processore disponibile, lo faccia eseguire da un processore diverso rispetto a quello su cui era stato eseguito in precedenza. Se il processo passa da un processore all'altro in questo modo, cosa che avveniva abbastanza di frequente con i kernel della serie 2.4.x, si ha l'effetto ping-pong.

Questo tipo di comportamento può generare dei seri problemi di prestazioni; infatti tutti i processori moderni utilizzano una memoria interna (la *cache*) contenente i dati più usati, che permette di evitare di eseguire un accesso (molto più lento) alla memoria principale sulla scheda madre. Chiaramente un processo sarà favorito se i suoi dati sono nella cache del processore, ma è ovvio che questo può essere vero solo per un processore alla volta, perché in presenza di più copie degli stessi dati su più processori, non si potrebbe determinare quale di questi ha la versione dei dati aggiornata rispetto alla memoria principale.

Questo comporta che quando un processore inserisce un dato nella sua cache, tutti gli altri processori che hanno lo stesso dato devono invalidarlo, e questa operazione è molto costosa in termini di prestazioni. Il problema diventa serio quando si verifica l'effetto ping-pong, in tal caso infatti un processo *rimbalza* continuamente da un processore all'altro e si ha una continua invalidazione della cache, che non diventa mai disponibile.

Per ovviare a questo tipo di problemi è nato il concetto di *affinità di processore* (o *CPU affinity*); la possibilità cioè di far sì che un processo possa essere assegnato per l'esecuzione sempre allo stesso processore. Lo *scheduler* dei kernel della serie 2.4.x aveva una scarsa *CPU affinity*, e l'effetto ping-pong era comune; con il nuovo *scheduler* dei kernel della 2.6.x questo problema è stato risolto ed esso cerca di mantenere il più possibile ciascun processo sullo stesso processore.

In certi casi però resta l'esigenza di poter essere sicuri che un processo sia sempre eseguito dallo stesso processore,⁴¹ e per poter risolvere questo tipo di problematiche nei nuovi kernel⁴²

⁴⁰è comunque possibile ripristinare un comportamento analogo al precedente scrivendo il valore 1 nel file `/proc/sys/kernel/sched_compat_yield`.

⁴¹quella che viene detta *hard CPU affinity*, in contrasto con quella fornita dallo *scheduler*, detta *soft CPU affinity*, che di norma indica solo una preferenza, non un requisito assoluto.

⁴²le due *system call* per la gestione della *CPU affinity* sono state introdotte nel kernel 2.5.8, e le corrispondenti funzioni di sistema nella *glibc* 2.3.

è stata introdotta l'opportuna infrastruttura ed una nuova *system call* che permette di impostare su quali processori far eseguire un determinato processo attraverso una *maschera di affinità*. La corrispondente funzione di sistema è `sched_setaffinity` ed il suo prototipo è:

```
#include <sched.h>
int sched_setaffinity(pid_t pid, size_t setsize, cpu_set_t *mask)
                                Imposta la maschera di affinità di un processo.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL il valore di `mask` contiene riferimenti a processori non esistenti nel sistema o a cui non è consentito l'accesso.

EPERM il processo non ha i privilegi sufficienti per eseguire l'operazione.

ESRCH il processo `pid` non esiste.

ed inoltre anche EFAULT nel suo significato generico.

Questa funzione e la corrispondente `sched_getaffinity` hanno una storia abbastanza complessa, la sottostante *system call* infatti prevede l'uso di due soli argomenti (per il `pid` e l'indicazione della maschera dei processori), che corrispondono al fatto che l'implementazione effettiva usa una semplice maschera binaria. Quando le funzioni vennero incluse nella *glibc* assunsero invece un prototipo simile a quello mostrato però con il secondo argomento di tipo `unsigned int`. A complicare la cosa si aggiunge il fatto che nella versione 2.3.3 della *glibc* detto argomento venne stato eliminato, per poi essere ripristinato nella versione 2.3.4 nella forma attuale.⁴³

La funzione imposta, con l'uso del valore contenuto all'indirizzo `mask`, l'insieme dei processori sui quali deve essere eseguito il processo identificato tramite il valore passato in `pid`. Come in precedenza il valore nullo di `pid` indica il processo corrente. Per poter utilizzare questa funzione sono richiesti i privilegi di amministratore (è necessaria la capacità `CAP_SYS_NICE`) altrimenti essa fallirà con un errore di `EPERM`. Una volta impostata una maschera di affinità, questa viene ereditata attraverso una `fork`, in questo modo diventa possibile legare automaticamente un gruppo di processi ad un singolo processore.

Nell'uso comune, almeno con i kernel successivi alla serie 2.6.x, utilizzare questa funzione non è necessario, in quanto è lo *scheduler* stesso che provvede a mantenere al meglio l'affinità di processore. Esistono però esigenze particolari, ad esempio quando un processo (o un gruppo di processi) è utilizzato per un compito importante (ad esempio per applicazioni *real-time* o la cui risposta è critica) e si vuole la massima velocità; con questa interfaccia diventa possibile selezionare gruppi di processori utilizzabili in maniera esclusiva. Lo stesso dicasi quando l'accesso a certe risorse (memoria o periferiche) può avere un costo diverso a seconda del processore, come avviene nelle architetture NUMA (*Non-Uniform Memory Access*).

Infine se un gruppo di processi accede alle stesse risorse condivise (ad esempio una applicazione con più *thread*) può avere senso usare lo stesso processore in modo da sfruttare meglio l'uso della sua cache; questo ovviamente riduce i benefici di un sistema multiprocessore nell'esecuzione contemporanea dei *thread*, ma in certi casi (quando i *thread* sono inerentemente serializzati nell'accesso ad una risorsa) possono esserci sufficienti vantaggi nell'evitare la perdita della cache da rendere conveniente l'uso dell'affinità di processore.

Dato che il numero di processori può variare a seconda delle architetture, per semplificare l'uso dell'argomento `mask` la *glibc* ha introdotto un apposito dato di tipo, `cpu_set_t`,⁴⁴ che

⁴³pertanto se la vostra pagina di manuale non è aggiornata, o usate quella particolare versione della *glibc*, potrete trovare indicazioni diverse, il prototipo illustrato è quello riportato nella versione corrente (maggio 2008) delle pagine di manuale e corrispondente alla definizione presente in `sched.h`.

⁴⁴questa è una estensione specifica della *glibc*, da attivare definendo la macro `_GNU_SOURCE`, non esiste infatti una standardizzazione per questo tipo di interfaccia e POSIX al momento non prevede nulla al riguardo.

permette di identificare un insieme di processori. Il dato è normalmente una maschera binaria: nei casi più comuni potrebbe bastare un intero a 32 bit, in cui ogni bit corrisponde ad un processore, ma oggi esistono architetture in cui questo numero può non essere sufficiente, e per questo è stato creato questo tipo opaco e una interfaccia di gestione che permette di usare a basso livello un tipo di dato qualunque rendendosi indipendenti dal numero di bit e dalla loro disposizione. Per questo le funzioni di libreria richiedono che oltre all'insieme di processori si indichi anche la dimensione dello stesso con l'argomento `setsize`, per il quale, se non si usa l'allocazione dinamica che vedremo a breve, è in genere sufficiente passare il valore `sizeof(cpu_set_t)`.

L'interfaccia di gestione degli insiemi di processori, oltre alla definizione del tipo `cpu_set_t`, prevede una serie di macro di preprocessore per la manipolazione degli stessi. Quelle di base, che consentono rispettivamente di svuotare un insieme, di aggiungere o togliere un processore o di verificare se esso è già presente in un insieme, sono le seguenti:

```
#include <sched.h>
void CPU_ZERO(cpu_set_t *set)           Inizializza un insieme di processori vuoto set.
void CPU_SET(int cpu, cpu_set_t *set)   Inserisce il processore cpu nell'insieme di processori set.
void CPU_CLR(int cpu, cpu_set_t *set)   Rimuove il processore cpu nell'insieme di processori set.
int CPU_ISSET(int cpu, cpu_set_t *set)  Controlla se il processore cpu è nell'insieme di processori set.
```

Queste macro che sono ispirate dalle analoghe usate per gli insiemi di *file descriptor* (vedi sez. 10.2.2) e sono state introdotte con la versione 2.3.3 della *glibc*. Tutte richiedono che si specifichi il numero di una CPU nell'argomento `cpu`, ed un insieme su cui operare. L'unica che ritorna un risultato è `CPU_ISSET`, che restituisce un intero da usare come valore logico (zero se la CPU non è presente, diverso da zero se è presente).

Si tenga presente che trattandosi di macro l'argomento `cpu` può essere valutato più volte. Questo significa ad esempio che non si può usare al suo posto una funzione o un'altra macro, altrimenti queste verrebbero eseguite più volte; l'argomento cioè non deve avere *effetti collaterali* (in gergo *side effects*).⁴⁵

Le CPU sono numerate da zero (che indica la prima disponibile) fino ad un numero massimo che dipende dall'architettura hardware. La costante `CPU_SETSIZE` indica il numero massimo di processori che possono far parte di un insieme (al momento vale sempre 1024), e costituisce un limite massimo al valore dell'argomento `cpu`. Dalla versione 2.6 della *glibc* alle precedenti macro è stata aggiunta, per contare il numero di processori in un insieme, l'ulteriore:

```
#include <sched.h>
int CPU_COUNT(cpu_set_t *set)           Conta il numero di processori presenti nell'insieme set.
```

A partire dalla versione 2.7 della *glibc* sono state introdotte altre macro che consentono ulteriori manipolazioni, in particolare si possono compiere delle operazioni logiche sugli insiemi di processori con:

⁴⁵nel linguaggio C si parla appunto di *side effects* quando si usano istruzioni la cui valutazione comporta effetti al di fuori dell'istruzione stessa, come il caso indicato in cui si passa una funzione ad una macro che usa l'argomento al suo interno più volte, o si scrivono espressioni come `a=a++` in cui non è chiaro se prima avvenga l'incremento e poi l'assegnazione, ed il cui risultato dipende dall'implementazione del compilatore.

```
#include <sched.h>
void CPU_AND(cpu_set_t *destset, cpu_set_t *srcset1, cpu_set_t *srcset2)
    Esegue l'AND logico di due insiemi di processori.
void CPU_OR(cpu_set_t *destset, cpu_set_t *srcset1, cpu_set_t *srcset2)
    Esegue l'OR logico di due insiemi di processori.
void CPU_XOR(cpu_set_t *destset, cpu_set_t *srcset1, cpu_set_t *srcset2)
    Esegue lo XOR logico di due insiemi di processori.
int CPU_EQUAL(cpu_set_t *set1, cpu_set_t *set2)
    Verifica se due insiemi di processori sono uguali.
```

Le prime tre macro richiedono due insiemi di partenza, `srcset1` e `srcset2` e forniscono in un terzo insieme `destset` (che può essere anche lo stesso di uno dei precedenti) il risultato della rispettiva operazione logica sui contenuti degli stessi. In sostanza con `CPU_AND` si otterrà come risultato l'insieme che contiene le CPU presenti in entrambi gli insiemi di partenza, con `CPU_OR` l'insieme che contiene le CPU presenti in uno qualunque dei due insiemi di partenza, e con `CPU_XOR` l'insieme che contiene le CPU presenti in uno solo dei due insiemi di partenza. Infine `CPU_EQUAL` confronta due insiemi ed è l'unica che restituisce un intero, da usare come valore logico che indica se sono identici o meno.

Inoltre, sempre a partire dalla versione 2.7 della *glibc*, è stata introdotta la possibilità di una allocazione dinamica degli insiemi di processori, per poterli avere di dimensioni corrispondenti al numero di CPU effettivamente in gioco, senza dover fare riferimento necessariamente alla precedente dimensione preimpostata di 1024. Per questo motivo sono state definite tre ulteriori macro, che consentono rispettivamente di allocare, disallocare ed ottenere la dimensione in byte di un insieme di processori:

```
#include <sched.h>
cpu_set_t * CPU_ALLOC(num_cpus)
    Alloca dinamicamente un insieme di processori di dimensione voluta.
void CPU_FREE(cpu_set_t *set)
    Disalloca un insieme di processori allocato dinamicamente.
size_t CPU_ALLOC_SIZE(num_cpus)
    Ritorna la dimensione di un insieme di processori allocato dinamicamente.
```

La prima macro, `CPU_ALLOC`, restituisce il puntatore ad un insieme di processori in grado di contenere almeno `num_cpus` che viene allocato dinamicamente. Ogni insieme così allocato dovrà essere disallocato con `CPU_FREE` passandogli un puntatore ottenuto da una precedente `CPU_ALLOC`. La terza macro, `CPU_ALLOC_SIZE`, consente di ottenere la dimensione in byte di un insieme allocato dinamicamente che contenga `num_cpus` processori.

Dato che le dimensioni effettive possono essere diverse le macro di gestione e manipolazione che abbiamo trattato in precedenza non si applicano agli insiemi allocati dinamicamente, per i quali dovranno essere state definite altrettante macro equivalenti contraddistinte dal suffisso `_S`, che effettuano le stesse operazioni, ma richiedono in più un argomento aggiuntivo `setsize` che deve essere assegnato al valore ottenuto con `CPU_ALLOC_SIZE`. Questo stesso valore deve essere usato per l'omonimo argomento delle funzioni `sched_setaffinity` o `sched_getaffinity` quando si vuole usare per l'argomento che indica la maschera di affinità un insieme di processori allocato dinamicamente.

A meno di non aver utilizzato `sched_setaffinity`, in condizioni ordinarie la maschera di affinità di un processo è preimpostata dal sistema in modo che esso possa essere eseguito su qualunque processore. Se ne può comunque ottenere il valore corrente usando la funzione di sistema `sched_getaffinity`, il cui prototipo è:

```
#include <sched.h>
int sched_getaffinity(pid_t pid, size_t setsize, cpu_set_t *mask)
    Legge la maschera di affinità di un processo.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `setsize` è più piccolo delle dimensioni della maschera di affinità usata dal kernel.

`ESRCH` il processo `pid` non esiste.

ed inoltre anche `EFAULT` nel suo significato generico.

La funzione restituirà all'indirizzo specificato da `mask` il valore della maschera di affinità del processo indicato dall'argomento `pid` (al solito un valore nullo indica il processo corrente) così da poterla riutilizzare per una successiva reimpostazione.

È chiaro che queste funzioni per la gestione dell'affinità hanno significato soltanto su un sistema multiprocessore, esse possono comunque essere utilizzate anche in un sistema con un processore singolo, nel qual caso però non avranno alcun risultato effettivo.

3.3.5 Le priorità per le operazioni di I/O

A lungo l'unica priorità usata per i processi è stata quella relativa all'assegnazione dell'uso del processore. Ma il processore non è l'unica risorsa che i processi devono contendersi, un'altra, altrettanto importante per le prestazioni, è quella dell'accesso a disco. Per questo motivo nello sviluppo del kernel sono stati introdotti diversi *I/O scheduler* in grado di distribuire in maniera opportuna questa risorsa ai vari processi.

Fino al kernel 2.6.17 era possibile soltanto differenziare le politiche generali di gestione, scegliendo di usare un diverso *I/O scheduler*. A partire da questa versione, con l'introduzione dello *scheduler* CFQ (*Completely Fair Queuing*) è divenuto possibile, qualora si usi questo *scheduler*, impostare anche delle diverse priorità di accesso per i singoli processi.⁴⁶

La scelta di uno *scheduler* di I/O si può fare in maniera generica per tutto il sistema all'avvio del kernel con il parametro di avvio `elevator`,⁴⁷ cui assegnare il nome dello *scheduler*, ma se ne può anche indicare uno specifico per l'accesso al singolo disco scrivendo nel file `/sys/block/<dev>/queue/scheduler` (dove `<dev>` è il nome del dispositivo associato al disco).

Gli *scheduler* disponibili sono mostrati dal contenuto dello stesso file che riporta fra parentesi quadre quello attivo, il default in tutti i kernel recenti è proprio il `cfq`, nome con cui si indica appunto lo *scheduler* CFQ, che supporta le priorità. Per i dettagli sulle caratteristiche specifiche degli altri *scheduler*, la cui discussione attiene a problematiche di ambito sistemistico, si consulti la documentazione nella directory `Documentation/block/` dei sorgenti del kernel.

Una volta che si sia impostato lo *scheduler* CFQ ci sono due specifiche *system call*, specifiche di Linux, che consentono di leggere ed impostare le priorità di I/O.⁴⁸ Dato che non esiste una interfaccia diretta nella *glibc* per queste due funzioni⁴⁹ occorrerà invocarle tramite la funzione `syscall` (come illustrato in sez. 2.1.2). Le due *system call* sono `ioprio_get` ed `ioprio_set`; i rispettivi prototipi sono:

```
#include <linux/ioprio.h>
int ioprio_get(int which, int who)
                                     Legge la priorità di I/O di un processo.
int ioprio_set(int which, int who, int ioprio)
                                     Imposta la priorità di I/O di un processo.
```

⁴⁶al momento (kernel 2.6.31), le priorità di I/O sono disponibili soltanto per questo *scheduler*.

⁴⁷per la trattazione dei parametri di avvio del kernel si rimanda al solito alla sez. 5.3 di [?].

⁴⁸se usate in corrispondenza ad uno *scheduler* diverso il loro utilizzo non avrà alcun effetto.

⁴⁹almeno al momento della scrittura di questa sezione, con la versione 2.11 della *glibc*.

Le funzioni ritornano rispettivamente un intero positivo o 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL i valori di `which` o di `ioprio` non sono validi.
 EPERM non si hanno i privilegi per eseguire l'impostazione (solo per `ioprio_set`).
 ESRCH non esiste un processo corrispondente alle indicazioni.

Le funzioni leggono o impostano la priorità di I/O sulla base dell'indicazione dei due argomenti `which` e `who` che hanno lo stesso significato già visto per gli omonimi argomenti di `getpriority` e `setpriority`. Anche in questo caso si deve specificare il valore di `which` tramite le opportune costanti riportate in tab. 3.11 che consentono di indicare un singolo processo, i processi di un *process group* (vedi sez. 8.1.2) o tutti i processi di un utente.

which	who	Significato
IPRIO_WHO_PROCESS	pid_t	processo
IPRIO_WHO_PRGR	pid_t	<i>process group</i>
IPRIO_WHO_USER	uid_t	utente

Tabella 3.11: Legenda del valore dell'argomento `which` e del tipo dell'argomento `who` delle funzioni `ioprio_get` e `ioprio_set` per le tre possibili scelte.

In caso di successo `ioprio_get` restituisce un intero positivo che esprime il valore della priorità di I/O, questo valore è una maschera binaria composta da due parti, una che esprime la classe di *scheduling* di I/O del processo, l'altra che esprime, quando la classe di *scheduling* lo prevede, la priorità del processo all'interno della classe stessa. Questo stesso formato viene utilizzato per indicare il valore della priorità da impostare con l'argomento `ioprio` di `ioprio_set`.

Macro	Significato
IOPRIO_PRIO_CLASS(value)	Dato il valore di una priorità come restituito da <code>ioprio_get</code> estrae il valore della classe.
IOPRIO_PRIO_DATA(value)	Dato il valore di una priorità come restituito da <code>ioprio_get</code> estrae il valore della priorità.
IOPRIO_PRIO_VALUE(class,prio)	Dato un valore di priorità ed una classe ottiene il valore numerico da passare a <code>ioprio_set</code> .

Tabella 3.12: Le macro per la gestione dei valori numerici .

Per la gestione dei valori che esprimono le priorità di I/O sono state definite delle opportune macro di preprocessore, riportate in tab. 3.12. I valori delle priorità si ottengono o si impostano usando queste macro. Le prime due si usano con il valore restituito da `ioprio_get` e per ottenere rispettivamente la classe di *scheduling*⁵⁰ e l'eventuale valore della priorità. La terza macro viene invece usata per creare un valore di priorità da usare come argomento di `ioprio_set` per eseguire una impostazione.

Le classi di *scheduling* previste dallo *scheduler* CFQ sono tre, e ricalcano tre diverse modalità di distribuzione delle risorse, analoghe a quelle già adottate anche nel funzionamento dello *scheduler* del processore. Ciascuna di esse è identificata tramite una opportuna costante, secondo quanto riportato in tab. 3.13.

La classe di priorità più bassa è `IOPRIO_CLASS_IDLE`; i processi in questa classe riescono ad accedere a disco soltanto quando nessun altro processo richiede l'accesso. Occorre pertanto usarla con molta attenzione, perché un processo in questa classe può venire completamente

⁵⁰restituita dalla macro con i valori di tab. 3.13.

Classe	Significato
IOPRIO_CLASS_RT	<i>Scheduling</i> di I/O <i>real-time</i> .
IOPRIO_CLASS_BE	<i>Scheduling</i> di I/O ordinario.
IOPRIO_CLASS_IDLE	<i>Scheduling</i> di I/O di priorità minima.

Tabella 3.13: Costanti che identificano le classi di *scheduling* di I/O.

bloccato quando ci sono altri processi in una qualunque delle altre due classi che stanno accedendo al disco. Quando si usa questa classe non ha senso indicare un valore di priorità, dato che in questo caso non esiste nessuna gerarchia e la priorità è identica, la minima possibile, per tutti i processi che la usano.

La seconda classe di priorità di I/O è IOPRIO_CLASS_BE (il nome sta per *best-effort*), che è quella usata ordinariamente da tutti processi. In questo caso esistono priorità diverse che consentono di assegnazione di una maggiore banda passante nell'accesso a disco ad un processo rispetto agli altri, con meccanismo simile a quello dei valori di *nice* in cui si evita che un processo a priorità più alta possa bloccare indefinitamente quelli a priorità più bassa. In questo caso però le diverse priorità sono soltanto otto, indicate da un valore numerico fra 0 e 7 e come per *nice* anche in questo caso un valore più basso indica una priorità maggiore.

Infine la classe di priorità di I/O *real-time* IOPRIO_CLASS_RT ricalca le omonime priorità di processore: un processo in questa classe ha sempre la precedenza nell'accesso a disco rispetto a tutti i processi delle altre classi e di un processo nella stessa classe ma con priorità inferiore, ed è pertanto in grado di bloccare completamente tutti gli altri. Anche in questo caso ci sono 8 priorità diverse con un valore numerico fra 0 e 7, con una priorità più elevata per valori più bassi.

In generale nel funzionamento ordinario la priorità di I/O di un processo viene impostata in maniera automatica nella classe IOPRIO_CLASS_BE con un valore ottenuto a partire dal corrispondente valore di *nice* tramite la formula: $prio = (nice + 20)/5$. Un utente ordinario può modificare con `ioprio_set` soltanto le priorità dei processi che gli appartengono,⁵¹ cioè quelli il cui *UID* reale corrisponde all'*UID* reale o effettivo del chiamante. Data la possibilità di ottenere un blocco totale del sistema, solo l'amministratore⁵² può impostare un processo ad una priorità di I/O nella classe IOPRIO_CLASS_RT, lo stesso privilegio era richiesto anche per la classe IOPRIO_CLASS_IDLE fino al kernel 2.6.24, ma dato che in questo caso non ci sono effetti sugli altri processi questo limite è stato rimosso a partire dal kernel 2.6.25.

3.4 Problematiche di programmazione *multitasking*

Benché i processi siano strutturati in modo da apparire il più possibile come indipendenti l'uno dall'altro, nella programmazione in un sistema *multitasking* occorre tenere conto di una serie di problematiche che normalmente non esistono quando si ha a che fare con un sistema in cui viene eseguito un solo programma alla volta.

Per questo motivo, essendo questo argomento di carattere generale, ci è parso opportuno introdurre sinteticamente queste problematiche, che ritroveremo a più riprese in capitoli successivi, in questa sezione conclusiva del capitolo in cui abbiamo affrontato la gestione dei processi, sottolineando come esse diventino cogenti quando invece si usano i *thread*.

⁵¹per la modifica delle priorità di altri processi occorrono privilegi amministrativi, ed in particolare la capacità CAP_SYS_NICE (vedi sez. 9.1.1).

⁵²o un processo con la capacità CAP_SYS_ADMIN (vedi sez. 9.1.1).

3.4.1 Le operazioni atomiche

La nozione di *operazione atomica* deriva dal significato greco della parola atomo, cioè indivisibile; si dice infatti che un'operazione è atomica quando si ha la certezza che, qualora essa venga effettuata, tutti i passaggi che devono essere compiuti per realizzarla verranno eseguiti senza possibilità di interruzione in una fase intermedia.

In un ambiente *multitasking* il concetto è essenziale, dato che un processo può essere interrotto in qualunque momento dal kernel che mette in esecuzione un altro processo o dalla ricezione di un segnale. Occorre pertanto essere accorti nei confronti delle possibili *race condition* (vedi sez. 3.4.2) derivanti da operazioni interrotte in una fase in cui non erano ancora state completate.

Nel caso dell'interazione fra processi la situazione è molto più semplice, ed occorre preoccuparsi della atomicità delle operazioni solo quando si ha a che fare con meccanismi di intercomunicazione (che esamineremo in dettaglio in cap. 11) o nelle operazioni con i file (vedremo alcuni esempi in sez. 5.2.1). In questi casi in genere l'uso delle appropriate funzioni di libreria per compiere le operazioni necessarie è garanzia sufficiente di atomicità in quanto le *system call* con cui esse sono realizzate non possono essere interrotte (o subire interferenze pericolose) da altri processi.

Nel caso dei segnali invece la situazione è molto più delicata, in quanto lo stesso processo, e pure alcune *system call*, possono essere interrotti in qualunque momento, e le operazioni di un eventuale *signal handler* sono compiute nello stesso spazio di indirizzi del processo. Per questo, anche il solo accesso o l'assegnazione di una variabile possono non essere più operazioni atomiche (torneremo su questi aspetti in sez. 7.4).

Qualora invece si usino i *thread*, in cui lo spazio degli indirizzi è condiviso, il problema è sempre presente, perché qualunque *thread* può interromperne un altro in qualunque momento e l'atomicità di qualunque operazione è messa in discussione, per cui l'assenza di eventuali *race condition* (vedi sez. 3.4.2) deve essere sempre verificata nei minimi dettagli.

In questo caso il sistema provvede un tipo di dato, il `sig_atomic_t`, il cui accesso è assicurato essere atomico. In pratica comunque si può assumere che, in ogni piattaforma su cui è implementato Linux, il tipo `int`, gli altri interi di dimensione inferiore ed i puntatori sono atomici. Non è affatto detto che lo stesso valga per interi di dimensioni maggiori (in cui l'accesso può comportare più istruzioni in assembler) o per le strutture di dati. In tutti questi casi è anche opportuno marcare come *volatile* le variabili che possono essere interessate ad accesso condiviso, onde evitare problemi con le ottimizzazioni del codice.

3.4.2 Le *race condition* ed i *deadlock*

Si definiscono *race condition* tutte quelle situazioni in cui processi diversi operano su una risorsa comune, ed in cui il risultato viene a dipendere dall'ordine in cui essi effettuano le loro operazioni. Il caso tipico è quello di un'operazione che viene eseguita da un processo in più passi, e può essere compromessa dall'intervento di un altro processo che accede alla stessa risorsa quando ancora non tutti i passi sono stati completati.

Dato che in un sistema *multitasking* ogni processo può essere interrotto in qualunque momento per farne subentrare un altro in esecuzione, niente può assicurare un preciso ordine di esecuzione fra processi diversi o che una sezione di un programma possa essere eseguita senza interruzioni da parte di altri. Queste situazioni comportano pertanto errori estremamente subdoli e difficili da tracciare, in quanto nella maggior parte dei casi tutto funzionerà regolarmente, e solo occasionalmente si avranno degli errori.

Per questo occorre essere ben consapevoli di queste problematiche, e del fatto che l'unico modo per evitarle è quello di riconoscerle come tali e prendere gli adeguati provvedimenti

per far sì che non si verifichino. Casi tipici di *race condition* si hanno quando diversi processi accedono allo stesso file, o nell'accesso a meccanismi di intercomunicazione come la memoria condivisa.

In questi casi, se non si dispone della possibilità di eseguire atomicamente le operazioni necessarie, occorre che quelle parti di codice in cui si compiono le operazioni sulle risorse condivise, quelle che in genere vengono denominate “sezioni critiche” del programma, siano opportunamente protette da meccanismi di sincronizzazione (vedremo alcune problematiche di questo tipo in cap. 11).

Nel caso dei *thread* invece la situazione è molto più delicata e sostanzialmente qualunque accesso in memoria (a buffer, variabili o altro) può essere soggetto a *race condition* dato potrebbe essere interrotto in qualunque momento da un altro *thread*. In tal caso occorre pianificare con estrema attenzione l'uso delle variabili ed utilizzare i vari meccanismi di sincronizzazione che anche in questo caso sono disponibili (torneremo su queste problematiche di questo tipo in cap. 12.4)

Un caso particolare di *race condition* sono poi i cosiddetti *deadlock* (traducibile in *condizione di stallo*), che particolarmente gravi in quanto comportano spesso il blocco completo di un servizio, e non il fallimento di una singola operazione. Per definizione un *deadlock* è una situazione in cui due o più processi non sono più in grado di proseguire perché ciascuno aspetta il risultato di una operazione che dovrebbe essere eseguita dall'altro.

L'esempio tipico di una situazione che può condurre ad un *deadlock* è quello in cui un flag di “occupazione” viene rilasciato da un evento asincrono (come un segnale o un altro processo) fra il momento in cui lo si è controllato (trovandolo occupato) e la successiva operazione di attesa per lo sblocco. In questo caso, dato che l'evento di sblocco del flag è avvenuto senza che ce ne accorgessimo proprio fra il controllo e la messa in attesa, quest'ultima diventerà perpetua (da cui il nome di *deadlock*).

In tutti questi casi è di fondamentale importanza il concetto di atomicità visto in sez. 3.4.1; questi problemi infatti possono essere risolti soltanto assicurandosi, quando essa sia richiesta, che sia possibile eseguire in maniera atomica le operazioni necessarie.

3.4.3 Le funzioni rientranti

Si dice *rientrante* una funzione che può essere interrotta in qualunque punto della sua esecuzione ed essere chiamata una seconda volta da un altro *thread* di esecuzione senza che questo comporti nessun problema nell'esecuzione della stessa. La problematica è comune nella programmazione con i *thread*, ma si hanno gli stessi problemi quando si vogliono chiamare delle funzioni all'interno dei gestori dei segnali.

Fintanto che una funzione opera soltanto con le variabili locali è rientrante; queste infatti vengono allocate nello *stack*, ed un'altra invocazione non fa altro che allocarne un'altra copia. Una funzione può non essere rientrante quando opera su memoria che non è nello *stack*. Ad esempio una funzione non è mai rientrante se usa una variabile globale o statica.

Nel caso invece la funzione operi su un oggetto allocato dinamicamente, la cosa viene a dipendere da come avvengono le operazioni: se l'oggetto è creato ogni volta e ritornato indietro la funzione può essere rientrante, se invece esso viene individuato dalla funzione stessa due chiamate alla stessa funzione potranno interferire quando entrambe faranno riferimento allo stesso oggetto. Allo stesso modo una funzione può non essere rientrante se usa e modifica un oggetto che le viene fornito dal chiamante: due chiamate possono interferire se viene passato lo stesso oggetto; in tutti questi casi occorre molta cura da parte del programmatore.

In genere le funzioni di libreria non sono rientranti, molte di esse ad esempio utilizzano variabili statiche, la *glibc* però mette a disposizione due macro di compilatore, `_REentrant` e

`_THREAD_SAFE`, la cui definizione attiva le versioni rientranti di varie funzioni di libreria, che sono identificate aggiungendo il suffisso `_r` al nome della versione normale.

Capitolo 4

La gestione di file e directory

In questo capitolo tratteremo in dettaglio le modalità con cui si gestiscono file e directory, iniziando da un approfondimento dell'architettura del sistema illustrata a grandi linee in sez. 1.2.1 ed illustrando le principali caratteristiche di un filesystem e le interfacce che consentono di controllarne il montaggio e lo smontaggio.

Esamineremo poi le funzioni di libreria che si usano per copiare, spostare e cambiare i nomi di file e directory e l'interfaccia che permette la manipolazione dei loro attributi. Tratteremo inoltre la struttura di base del sistema delle protezioni e del controllo dell'accesso ai file e le successive estensioni (*Extended Attributes*, ACL, quote disco). Tutto quello che riguarda invece la gestione dell'I/O sui file è lasciato al capitolo successivo.

4.1 L'architettura della gestione dei file

In questa sezione tratteremo con maggiori dettagli rispetto a quanto visto in sez. 1.2.1 il *Virtual File System* di Linux e come il kernel può gestire diversi tipi di filesystem, descrivendo prima le caratteristiche generali di un filesystem di un sistema unix-like, per poi fare una panoramica sul filesystem tradizionalmente più usato con Linux, l'ext2 ed i suoi successori.

4.1.1 Il funzionamento del *Virtual File System* di Linux

Come illustrato brevemente in sez. 1.2.1 in Linux il concetto di *everything is a file* è stato implementato attraverso il *Virtual File System*, la cui struttura generale è illustrata in fig. 1.2. Il VFS definisce un insieme di funzioni che tutti i filesystem devono implementare per l'accesso ai file che contengono e l'interfaccia che consente di eseguire l'I/O sui file, che questi siano di dati o dispositivi.

L'interfaccia fornita dal VFS comprende in sostanza tutte le funzioni che riguardano i file, le operazioni implementate dal VFS sono realizzate con una astrazione che prevede quattro tipi di oggetti strettamente correlati: i filesystem, le *dentry*, gli *inode* ed i file. A questi oggetti corrispondono una serie di apposite strutture definite dal kernel che contengono come campi le funzioni di gestione e realizzano l'infrastruttura del VFS. L'interfaccia è molto complessa, ne faremo pertanto una trattazione estremamente semplificata che consenta di comprenderne i principi di funzionamento.

Il VFS usa una tabella mantenuta dal kernel che contiene il nome di ciascun filesystem supportato, quando si vuole inserire il supporto di un nuovo filesystem tutto quello che occorre è chiamare la funzione `register_filesystem` passando come argomento la struttura

`file_system_type` (la cui definizione è riportata in fig. 4.1) relativa a quel filesystem. Questa verrà inserita nella tabella, ed il nuovo filesystem comparirà in `/proc/filesystems`.

```

struct file_system_type {
    const char *name;
    int fs_flags;
    int (*get_sb) (struct file_system_type *, int,
                  const char *, void *, struct vfsmount *);
    struct dentry *(*mount) (struct file_system_type *, int,
                             const char *, void *);
    void (*kill_sb) (struct super_block *);
    struct module *owner;
    struct file_system_type * next;
    struct list_head fs_supers;
    ...
};

```

Figura 4.1: Estratto della struttura `file_system_type` usata dal VFS (da `include/linux/fs.h`).

La struttura `file_system_type`, oltre ad una serie di dati interni, come il nome del tipo di filesystem nel campo `name`,¹ contiene i riferimenti alle funzioni di base che consentono l'utilizzo di quel filesystem. In particolare la funzione `mount` del quarto campo è quella che verrà invocata tutte le volte che si dovrà effettuare il montaggio di un filesystem di quel tipo. Per ogni nuovo filesystem si dovrà allocare una di queste strutture ed inizializzare i relativi campi con i dati specifici di quel filesystem, ed in particolare si dovrà creare anche la relativa versione della funzione `mount`.

Come illustrato in fig. 4.1 questa funzione restituisce una *dentry*, abbreviazione che sta per *directory entry*. Le *dentry* sono gli oggetti che il kernel usa per eseguire la *pathname resolution*, ciascuna di esse corrisponde ad un *pathname* e contiene il riferimento ad un *inode*, che come vedremo a breve è l'oggetto usato dal kernel per identificare un file.² La *dentry* ottenuta dalla chiamata alla funzione `mount` sarà inserita in corrispondenza al *pathname* della directory in cui il filesystem è stato montato.

Le *dentry* sono oggetti del VFS che vivono esclusivamente in memoria, nella cosiddetta *directory entry cache* (spesso chiamata in breve *dcache*). Ogni volta che una *system call* specifica un *pathname* viene effettuata una ricerca nella *dcache* per ottenere immediatamente la *dentry* corrispondente,³ che a sua volta ci darà, tramite l'*inode*, il riferimento al file.

Dato che normalmente non è possibile mantenere nella *dcache* le informazioni relative a tutto l'albero dei file la procedura della *pathname resolution* richiede un meccanismo con cui riempire gli eventuali vuoti. Il meccanismo prevede che tutte le volte che si arriva ad una *dentry* mancante venga invocata la funzione `lookup` dell'*inode* associato alla *dentry* precedente nella risoluzione del *pathname*,⁴ il cui scopo è risolvere il nome mancante e fornire la sua *dentry* che a questo punto verrà inserita nella cache.

Dato che tutte le volte che si monta un filesystem la funzione `mount` (vedi sez. 4.1.4) della corrispondente `file_system_type` inserisce la *dentry* iniziale nel *mount point* dello stesso, si avrà comunque un punto di partenza. Inoltre essendo questa *dentry* relativa a quel tipo di

¹quello che viene riportato in `/proc/filesystems` e che viene usato come valore del parametro dell'opzione `-t` del comando `mount` che indica il tipo di filesystem.

²in questo caso si parla di file come di un qualunque oggetto generico che sta sul filesystem e non dell'oggetto file del VFS cui accennavamo prima.

³il buon funzionamento della *dcache* è in effetti di una delle parti più critiche per le prestazioni del sistema.

⁴che a questo punto è una directory, per cui si può cercare al suo interno il nome di un file.

filesystem essa farà riferimento ad un *inode* di quel filesystem, e come vedremo questo farà sì che venga eseguita una *lookup* adatta per effettuare la risoluzione dei nomi per quel filesystem.

L'oggetto più importante per il funzionamento del VFS è probabilmente l'*inode*, ma con questo nome si può fare riferimento a due cose diverse. La prima è la struttura su disco (su cui torneremo anche in sez. 4.1.2) che fa parte della organizzazione dei dati realizzata dal filesystem e che contiene le informazioni relative alle proprietà (i cosiddetti *metadati*) di ogni oggetto presente su di esso (si intende al solito uno qualunque dei tipi di file di tab. 1.1).

La seconda è la corrispondente struttura *inode*, della cui definizione si è riportato un estratto in fig. 4.2.⁵ Questa struttura viene mantenuta in memoria ed è a questa che facevamo riferimento quando parlavamo dell'*inode* associato a ciascuna *dentry*. Nella struttura in memoria sono presenti gli stessi *metadati* memorizzati su disco, che vengono letti quando questa struttura viene allocata e trascritti all'indietro se modificati.

```

struct inode {
    ...
    unsigned long      i_ino;
    atomic_t           i_count;
    unsigned int        i_nlink;
    uid_t              i_uid;
    gid_t              i_gid;
    dev_t              i_rdev;
    unsigned int        i_blkbits;
    u64                 i_version;
    loff_t              i_size;
    struct timespec     i_atime;
    struct timespec     i_mtime;
    struct timespec     i_ctime;
    ...
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    ...
};

```

Figura 4.2: Estratto della struttura *inode* del kernel (da `include/linux/fs.h`).

Il fatto che la struttura *inode* sia mantenuta in memoria, direttamente associata ad una *dentry*, rende sostanzialmente immediate le operazioni che devono semplicemente effettuare un accesso ai dati in essa contenuti: è così ad esempio che viene realizzata la *system call* *stat* che vedremo in sez. 4.3.1. Rispetto ai dati salvati sul disco questa struttura contiene però anche quanto necessario alla implementazione del VFS, ed in particolare è importante il campo *i_op* che, come illustrato in fig. 4.2, contiene il puntatore ad una struttura di tipo *inode_operation*, la cui definizione si può trovare nel file `include/kernel/fs.h` dei sorgenti del kernel.

Questa struttura non è altro che una tabella di funzioni, ogni suo membro cioè è un puntatore ad una funzione e, come suggerisce il nome della struttura stessa, queste funzioni sono quelle che definiscono le operazioni che il VFS può compiere su un *inode*. Si sono riportate in tab. 4.1 le più rilevanti.

Possiamo notare come molte di queste funzioni abbiano nomi sostanzialmente identici alle varie *system call* con le quali si gestiscono file e directory, che tratteremo nel resto del capitolo. Quello che succede è che tutte le volte che deve essere eseguita una *system call*, o una

⁵l'estratto fa riferimento alla versione del kernel 2.6.37.

Funzione	Operazione
create	Chiamata per creare un nuovo file (vedi sez. 5.1.2).
link	Crea un <i>hard link</i> (vedi sez. 4.2.1).
unlink	Cancella un <i>hard link</i> (vedi sez. 4.2.1).
symlink	Crea un collegamento simbolico (vedi sez. 4.2.1).
mkdir	Crea una directory (vedi sez. 4.2.2).
rmdir	Rimuove una directory (vedi sez. 4.2.2).
mknod	Crea un file speciale (vedi sez. 4.2.5).
rename	Cambia il nome di un file (vedi sez. 4.2.1).
lookup	Risolve il nome di un file.

Tabella 4.1: Le principali operazioni sugli *inode* definite tramite *inode_operation*.

qualunque altra operazione su un *inode* (come *lookup*) il VFS andrà ad utilizzare la funzione corrispondente attraverso il puntatore *i_op*.

Sarà allora sufficiente che nella realizzazione di un filesystem si crei una implementazione di queste funzioni per quel filesystem e si allochi una opportuna istanza di *inode_operation* contenente i puntatori a dette funzioni. A quel punto le strutture *inode* usate per gli oggetti di quel filesystem otterranno il puntatore alla relativa istanza di *inode_operation* e verranno automaticamente usate le funzioni corrette.

Si noti però come in tab. 4.1 non sia presente la funzione *open* che invece è citata in tab. 4.2.⁶ Questo avviene perché su Linux l'apertura di un file richiede comunque un'altra operazione che mette in gioco l'omonimo oggetto del VFS: l'allocazione di una struttura di tipo *file* che viene associata ad ogni file aperto nel sistema. I motivi per cui viene usata una struttura a parte sono diversi, anzitutto, come illustrato in sez. 5.1.1, questa è necessaria per le operazioni eseguite dai processi con l'interfaccia dei file descriptor. Ogni processo infatti mantiene il riferimento ad una struttura *file* per ogni file che ha aperto, ed è tramite essa che esegue le operazioni di I/O. Inoltre il kernel mantiene un elenco di tutti i file aperti nella *file table* (torneremo su questo in sez. 5.1.1).

Inoltre se le operazioni relative agli *inode* fanno riferimento ad oggetti posti all'interno di un filesystem e vi si applicano quindi le funzioni fornite nell'implementazione di quest'ultimo, quando si apre un file questo può essere anche un file di dispositivo, ed in questo caso il VFS invece di usare le operazioni fornite dal filesystem (come farebbe per un file di dati) dovrà invece ricorrere a quelle fornite dal driver del dispositivo.

```

struct file {
    ...
    struct path          f_path;
    const struct file_operations *f_op;
    ...
    atomic_long_t        f_count;
    unsigned int         f_flags;
    fmode_t              f_mode;
    loff_t               f_pos;
    ...
};

```

Figura 4.3: Estratto della struttura *file* del kernel (da *include/linux/fs.h*).

⁶essa può essere comunque invocata dato che nella struttura *inode* è presente anche il puntatore *i_fop* alla struttura *file_operation* che fornisce detta funzione.

Come si può notare dall'estratto di fig. 4.3, la struttura `file` contiene, oltre ad alcune informazioni usate dall'interfaccia dei file descriptor il cui significato emergerà più avanti (vedi sez. 5.1), il puntatore `f_op` ad una struttura `file_operation`. Questa è l'analoga per i file di `inode_operation`, e definisce le operazioni generiche fornite dal VFS per i file. Si sono riportate in tab. 4.2 le più significative.

Funzione	Operazione
<code>open</code>	Apri il file (vedi sez. 5.1.2).
<code>read</code>	Legge dal file (vedi sez. 5.1.4).
<code>write</code>	Scrivi sul file (vedi sez. 5.1.5).
<code>lseek</code>	Sposta la posizione corrente sul file (vedi sez. 5.1.3).
<code>ioctl</code>	Accede alle operazioni di controllo (vedi sez. 5.2.5).
<code>readdir</code>	Legge il contenuto di una directory (vedi sez. 4.2.3).
<code>poll</code>	Usata nell'I/O multiplexing (vedi sez. 10.2).
<code>mmap</code>	Mappa il file in memoria (vedi sez. 10.4.1).
<code>release</code>	Chiamata quando l'ultimo riferimento a un file aperto è chiuso.
<code>fsync</code>	Sincronizza il contenuto del file (vedi sez. 5.2.3).
<code>fsync</code>	Abilita l'I/O asincrono (vedi sez. 10.3.3) sul file.

Tabella 4.2: Operazioni sui file definite tramite `file_operation`.

Anche in questo caso tutte le volte che deve essere eseguita una *system call* o una qualunque altra operazione sul file il VFS andrà ad utilizzare la funzione corrispondente attraverso il puntatore `f_op`. Dato che è cura del VFS quando crea la struttura all'apertura del file assegnare a `f_op` il puntatore alla versione di `file_operation` corretta per quel file, sarà possibile scrivere allo stesso modo sulla porta seriale come su un normale file di dati, e lavorare sui file allo stesso modo indipendentemente dal filesystem.

Il VFS realizza la quasi totalità delle operazioni relative ai file grazie alle funzioni presenti nelle due strutture `inode_operation` e `file_operation`. Ovviamente non è detto che tutte le operazioni possibili siano poi disponibili in tutti i casi, ad esempio `lseek` non sarà presente per un dispositivo come la porta seriale o per una *fifo*, mentre sui file del filesystem `vfat` non saranno disponibili i permessi, ma resta il fatto che grazie al VFS le *system call* per le operazioni sui file possono restare sempre le stesse nonostante le enormi differenze che possono esserci negli oggetti a cui si applicano.

4.1.2 Il funzionamento di un filesystem Unix

Come già accennato in sez. 1.2.1 Linux (ed ogni sistema unix-like) organizza i dati che tiene su disco attraverso l'uso di un filesystem. Una delle caratteristiche di Linux rispetto agli altri Unix è quella di poter supportare, grazie al VFS, una enorme quantità di filesystem diversi, ognuno dei quali avrà una sua particolare struttura e funzionalità proprie. Per questo non entreremo nei dettagli di un filesystem specifico, ma daremo una descrizione a grandi linee che si adatta alle caratteristiche comuni di qualunque filesystem di un sistema unix-like.

Una possibile strutturazione dell'informazione su un disco è riportata in fig. 4.4, dove si hanno tre filesystem su tre partizioni. In essa per semplicità si è fatto riferimento alla struttura del filesystem `ext2`, che prevede una suddivisione dei dati in *block group*. All'interno di ciascun *block group* viene anzitutto replicato il cosiddetto *superblock*, (la struttura che contiene l'indice iniziale del filesystem e che consente di accedere a tutti i dati sottostanti) e creata una opportuna suddivisione dei dati e delle informazioni per accedere agli stessi. Sulle caratteristiche di `ext2` e derivati torneremo in sez. 4.1.3.

È comunque caratteristica comune di tutti i filesystem per Unix, indipendentemente da come poi viene strutturata nei dettagli questa informazione, prevedere la presenza di due

tipi di risorse: gli *inode*, cui abbiamo già accennato in sez. 4.1.1, che sono le strutture che identificano i singoli oggetti sul filesystem, e i blocchi, che invece attengono allo spazio disco che viene messo a disposizione per i dati in essi contenuti.

Figura 4.4: Organizzazione dello spazio su un disco in partizioni e filesystem.

Se si va ad esaminare con maggiore dettaglio la strutturazione dell'informazione all'interno del filesystem *ext2*, tralasciando i dettagli relativi al funzionamento del filesystem stesso come la strutturazione in gruppi dei blocchi, il *superblock* e tutti i dati di gestione possiamo esemplificare la situazione con uno schema come quello esposto in fig. 4.5.

Figura 4.5: Strutturazione dei dati all'interno di un filesystem.

Da fig. 4.5 si evidenziano alcune delle caratteristiche di base di un filesystem, che restano le stesse anche su filesystem la cui organizzazione dei dati è totalmente diversa da quella illustrata, e sulle quali è bene porre attenzione visto che sono fondamentali per capire il funzionamento delle funzioni che manipolano i file e le directory che tratteremo nel prosieguo del capitolo. In particolare è opportuno tenere sempre presente che:

1. L'*inode* contiene i cosiddetti *metadati*, vale dire le informazioni riguardanti le proprietà del file come oggetto del filesystem: il tipo di file, i permessi di accesso, le dimensioni, i puntatori ai blocchi fisici che contengono i dati e così via. Le informazioni che la funzione *stat* (vedi sez. 4.3.1) fornisce provengono dall'*inode*. Dentro una directory si troverà solo il nome del file e il numero dell'*inode* ad esso associato; il nome non è una proprietà del file e non viene mantenuto nell'*inode*. Da qui in poi chiameremo il nome del file contenuto in una directory "voce", come traduzione della nomenclatura inglese *directory entry* che non useremo per evitare confusione con le *dentry* del kernel viste in sez. 4.1.1.
2. Come mostrato in fig. 4.5 per i file *macro.tex* e *gatil_macro.tex*, ci possono avere più voci che fanno riferimento allo stesso *inode*. Fra le proprietà di un file mantenute nell'*inode* c'è anche il contatore con il numero di riferimenti che sono stati fatti ad esso, il cosiddetto *link count*.⁷ Solo quando questo contatore si annulla i dati del file possono essere effettivamente rimossi dal disco. Per questo la funzione per cancellare un file si chiama *unlink* (vedi sez. 4.2.1), ed in realtà non cancella affatto i dati del file, ma si limita ad eliminare la relativa voce da una directory e decrementare il numero di riferimenti nell'*inode*.
3. All'interno di ogni filesystem ogni *inode* è identificato da un numero univoco. Il numero di *inode* associato ad una voce in una directory si riferisce ad questo numero e non ci può essere una directory che contiene riferimenti ad *inode* relativi ad altri filesystem. Questa è la ragione che limita l'uso del comando *ln*, che crea una nuova voce per un file esistente con la funzione *link* (vedi sez. 4.2.1), a operare su file nel filesystem corrente.
4. Quando si cambia nome ad un file senza cambiare filesystem il contenuto del file non viene spostato fisicamente, viene semplicemente creata una nuova voce per l'*inode* in questione e rimossa la precedente, questa è la modalità in cui opera normalmente il comando *mv* attraverso la funzione *rename* (vedi sez. 4.2.1). Questa operazione non modifica minimamente neanche l'*inode* del file, dato che non si opera sul file ma sulla directory che lo contiene.

⁷mantenuto anche nel campo *i_nlink* della struttura *inode* di fig. 4.2.

5. Gli *inode* dei file, che contengono i *metadati*, ed i blocchi di spazio disco, che contengono i dati, sono risorse indipendenti ed in genere vengono gestite come tali anche dai diversi filesystem; è pertanto possibile esaurire sia lo spazio disco (il caso più comune) che lo spazio per gli *inode*. Nel primo caso non sarà possibile allocare ulteriore spazio, ma si potranno creare file (vuoti), nel secondo non si potranno creare nuovi file, ma si potranno estendere quelli che ci sono.⁸

Figura 4.6: Organizzazione dei *link* per le directory.

Infine tenga presente che, essendo file pure loro, il numero di riferimenti esiste anche per le directory. Per questo se a partire dalla situazione mostrata in fig. 4.5 creiamo una nuova directory *img* nella directory *gapil*, avremo una situazione come quella illustrata in fig. 4.6.

La nuova directory avrà un numero di riferimenti pari a due, in quanto è referenziata dalla directory da cui si era partiti (in cui è inserita la nuova voce che fa riferimento a *img*) e dalla voce interna “.” che è presente in ogni directory. Questo è il valore che si troverà sempre per ogni directory che non contenga a sua volta altre directory. Al contempo, la directory da cui si era partiti avrà un numero di riferimenti di almeno tre, in quanto adesso sarà referenziata anche dalla voce “.” di *img*. L’aggiunta di una sottodirectory fa cioè crescere di uno il *link count* della directory genitrice.

4.1.3 Alcuni dettagli sul filesystem *ext2* e successori

Benché non esista “il” filesystem di Linux, dato che esiste un supporto nativo di diversi filesystem che sono in uso da anni, quello che gli avvicina di più è la famiglia di filesystem evolutasi a partire dal *second extended filesystem*, o *ext2*. Il filesystem *ext2* ha subito un grande sviluppo e diverse evoluzioni, fra cui l’aggiunta del *journaling* con il passaggio ad *ext3*, che probabilmente è ancora il filesystem più diffuso, ed una serie di ulteriori miglioramenti con il successivo *ext4*. In futuro è previsto che questo debba essere sostituito da un filesystem completamente diverso, *Btrfs*, che dovrebbe diventare il filesystem standard di Linux, ma questo al momento è ancora in fase di sviluppo.⁹

Il filesystem *ext2* nasce come filesystem nativo per Linux a partire dalle prime versioni del kernel e supporta tutte le caratteristiche di un filesystem standard Unix: è in grado di gestire nomi di file lunghi (256 caratteri, estensibili a 1012) e supporta una dimensione massima dei file fino a 4 Tb. I successivi filesystem *ext3* ed *ext4* sono evoluzioni di questo filesystem, e sia pure con molti miglioramenti ed estensioni significative ne mantengono le caratteristiche fondamentali.

Oltre alle caratteristiche standard, *ext2* fornisce alcune estensioni che non sono presenti su un classico filesystem di tipo Unix; le principali sono le seguenti:

- gli attributi estesi (vedi sez. 4.5.1) che consentono di estendere le informazioni salvabili come metadati e le ACL (vedi sez. 4.5.2) che consentono di estendere il modello tradizionale dei permessi sui file.
- sono supportate entrambe le semantiche di BSD e SVr4 come opzioni di montaggio. La semantica BSD comporta che i file in una directory sono creati con lo stesso identificatore di gruppo della directory che li contiene. La semantica SVr4 comporta che i file vengono

⁸questo comportamento non è generale, alcuni filesystem più sofisticati possono evitare il problema dell’esaurimento degli *inode* riallocando lo spazio disco libero per i blocchi.

⁹si fa riferimento al momento dell’ultima revisione di questo paragrafo, l’inizio del 2012.

creati con l'identificatore del gruppo primario del processo, eccetto il caso in cui la directory ha il bit di *sgid* impostato (per una descrizione dettagliata del significato di questi termini si veda sez. 4.4), nel qual caso file e subdirectory ereditano sia il *GID* che lo *sgid*.

- l'amministratore può scegliere la dimensione dei blocchi del filesystem in fase di creazione, a seconda delle sue esigenze: blocchi più grandi permettono un accesso più veloce, ma sprecano più spazio disco.
- il filesystem implementa collegamenti simbolici veloci, in cui il nome del file non è salvato su un blocco, ma tenuto all'interno dell'*inode* (evitando letture multiple e spreco di spazio), non tutti i nomi però possono essere gestiti così per limiti di spazio (il limite è 60 caratteri).
- vengono supportati i cosiddetti *file attributes* (vedi sez. 4.4.1) che attivano comportamenti specifici per i file su cui vengono attivati come marcarli come immutabili (che possono cioè essere soltanto letti) per la protezione di file di configurazione sensibili, o come *append-only* (che possono essere aperti in scrittura solo per aggiungere dati) per la protezione dei file di log.

La struttura di *ext2* è stata ispirata a quella del filesystem di BSD: un filesystem è composto da un insieme di blocchi, la struttura generale è quella riportata in fig. 4.5, in cui la partizione è divisa in gruppi di blocchi.

Ciascun gruppo di blocchi contiene una copia delle informazioni essenziali del filesystem (i *superblock* sono quindi ridondati) per una maggiore affidabilità e possibilità di recupero in caso di corruzione del *superblock* principale. L'utilizzo di raggruppamenti di blocchi ha inoltre degli effetti positivi nelle prestazioni dato che viene ridotta la distanza fra i dati e la tabella degli *inode*.

Figura 4.7: Struttura delle directory nel *second extended filesystem*.

Le directory sono implementate come una *linked list* con voci di dimensione variabile. Ciascuna voce della lista contiene il numero di *inode*, la sua lunghezza, il nome del file e la sua lunghezza, secondo lo schema in fig. 4.7; in questo modo è possibile implementare nomi per i file anche molto lunghi (fino a 1024 caratteri) senza sprecare spazio disco.

Con l'introduzione del filesystem *ext3* sono state introdotte diverse modifiche strutturali, la principale di queste è quella che *ext3* è un filesystem *journaled*, è cioè in grado di eseguire una registrazione delle operazioni di scrittura su un giornale (uno speciale file interno) in modo da poter garantire il ripristino della coerenza dei dati del filesystem¹⁰ in brevissimo tempo in caso di interruzione improvvisa della corrente o di crollo del sistema che abbia causato una interruzione della scrittura dei dati sul disco.

Oltre a questo *ext3* introduce ulteriori modifiche volte a migliorare sia le prestazioni che la semplicità di gestione del filesystem, in particolare per le directory si è passato all'uso di alberi binari con indicizzazione tramite *hash* al posto delle *linked list* che abbiamo illustrato, ottenendo un forte guadagno di prestazioni in caso di directory contenenti un gran numero di file.

¹⁰si noti bene che si è parlato di dati *del* filesystem, non di dati *nel* filesystem, quello di cui viene garantito un veloce ripristino è relativo ai dati della struttura interna del filesystem, non di eventuali dati contenuti nei file che potrebbero essere stati persi.

4.1.4 La gestione dell'uso dei filesystem

Come accennato in sez. 1.2.1 per poter accedere ai file occorre rendere disponibile al sistema il filesystem su cui essi sono memorizzati. L'operazione di attivazione del filesystem è chiamata *montaggio* e per far questo in Linux si usa la funzione di sistema `mount`, il cui prototipo è:¹¹

```
#include <sys/mount.h>
mount(const char *source, const char *target, const char *filesystemtype,
      unsigned long mountflags, const void *data)
```

Monta un filesystem.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES	non si ha il permesso di accesso su uno dei componenti del <i>pathname</i> , o si è cercato di montare un filesystem disponibile in sola lettura senza aver specificato <code>MS_RDONLY</code> o il device <i>source</i> è su un filesystem montato con l'opzione <code>MS_NODEV</code> .
EBUSY	<i>source</i> è già montato, o non può essere rimontato in sola lettura perché ci sono ancora file aperti in scrittura, o non può essere montato su <i>target</i> perché la directory è ancora in uso.
EINVAL	il dispositivo <i>source</i> presenta un <i>superblock</i> non valido, o si è cercato di rimontare un filesystem non ancora montato, o di montarlo senza che <i>target</i> sia un <i>mount point</i> o di spostarlo quando <i>target</i> non è un <i>mount point</i> o è la radice o si è usato un valore di <i>mountflags</i> non valido.
ELOOP	si è cercato di spostare un <i>mount point</i> su una sottodirectory di <i>source</i> o si sono incontrati troppi collegamenti simbolici nella risoluzione di un nome.
EMFILE	in caso di filesystem virtuale, la tabella dei dispositivi fittizi (chiamati <i>dummy</i> nella documentazione inglese) è piena.
ENODEV	il tipo <i>filesystemtype</i> non esiste o non è configurato nel kernel.
ENOTBLK	non si è usato un <i>block device</i> per <i>source</i> quando era richiesto.
ENXIO	il <i>major number</i> del dispositivo <i>source</i> è sbagliato.
EPERM	il processo non ha i privilegi di amministratore.

ed inoltre `EFAULT`, `ENOMEM`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR` nel loro significato generico.

L'uso più comune della funzione è quello di montare sulla directory indicata da **target**, detta *mount point*, il filesystem contenuto nel file di dispositivo indicato da **source**. In entrambi i casi, come daremo per assunto da qui in avanti tutte le volte che si parla di directory o file nel passaggio di un argomento di una funzione, si intende che questi devono essere indicati con la stringa contenente il loro *pathname*.

Normalmente un filesystem è contenuto su un disco o una partizione, ma come illustrato in sez. 4.1.1 la struttura del *Virtual File System* è estremamente flessibile e può essere usata anche per oggetti diversi da un disco. Ad esempio usando il *loop device* si può montare un file qualunque (come l'immagine di un CD-ROM o di un floppy) che contiene l'immagine di un filesystem, inoltre alcuni tipi di filesystem, come `proc` o `sysfs` sono virtuali e non hanno un supporto che ne contenga i dati che sono generati al volo dal kernel ad ogni lettura, e inviati al kernel ad ogni scrittura (costituiscono quindi un meccanismo di comunicazione, attraverso l'ordinaria interfaccia dei file, con il kernel).

Il tipo di filesystem che si vuole montare è specificato dall'argomento **filesystemtype**, che deve essere una delle stringhe riportate nel file `/proc/filesystems` che, come accennato in sez. 4.1.1, contiene l'elenco dei filesystem supportati dal kernel. Nel caso si sia indicato un filesystem virtuale, che non è associato a nessun file di dispositivo, il contenuto di **source** viene ignorato.

¹¹la funzione è una versione specifica di Linux che usa la omonima *system call* e non è portabile.

L'argomento **data** viene usato per passare le impostazioni relative alle caratteristiche specifiche di ciascun filesystem. Si tratta di una stringa di parole chiave (separate da virgole e senza spazi) che indicano le cosiddette “*opzioni*” del filesystem che devono essere impostate; in genere viene usato direttamente il contenuto del parametro dell'opzione **-o** del comando **mount**. I valori utilizzabili dipendono dal tipo di filesystem e ciascuno ha i suoi, pertanto si rimanda alla documentazione della pagina di manuale di questo comando e dei singoli filesystem.

Dopo l'esecuzione della funzione il contenuto del filesystem viene reso disponibile nella directory specificata come *mount point* ed il precedente contenuto di detta directory viene mascherato dal contenuto della directory radice del filesystem montato. Fino ai kernel della serie 2.2.x non era possibile montare un filesystem se un *mount point* era già in uso, coi kernel successivi è possibile montare più filesystem sullo stesso *mount point* impilandoli l'uno sull'altro, anche in questo caso vale quanto appena detto, e solo il contenuto dell'ultimo filesystem montato sarà visibile, mascherando quelli sottostanti.

In realtà quella di montare un filesystem è solo una delle operazioni che si possono effettuare con **mount**, la funzione infatti è dedicata a tutte le operazioni relative alla gestione del montaggio dei filesystem e dei *mount point*. Ad esempio fin dalle sue origini poteva essere utilizzata per effettuare il rimontaggio di un filesystem con opzioni diverse, ed a partire dal kernel 2.4.x è divenuto possibile usarla per spostare atomicamente un *mount point* da una directory ad un'altra, per montare lo stesso filesystem in diversi *mount point*, per montare una directory su un'altra (il cosiddetto *bind mount*).

Il tipo di operazione compiuto da **mount** viene stabilito in base al valore dell'argomento **mountflags**, che oltre alla selezione del tipo di operazione da compiere, consente anche di indicare alcune opzioni generiche valide per qualunque filesystem.¹² Il valore dell'argomento deve essere espresso come maschera binaria e i vari bit che lo compongono, detti anche *mount flags*, devono essere impostati con un OR aritmetico dei valori dalle opportune costanti che illustreremo a breve.

In Linux **mountflags** deve essere un intero a 32 bit; fino ai kernel della serie 2.2.x i 16 più significativi avevano un valore riservato che doveva essere specificato obbligatoriamente,¹³ e si potevano usare solo i 16 meno significativi. Oggi invece, con un numero di opzioni superiore, sono utilizzati tutti e 32 i bit, ma qualora nei 16 più significativi sia presente detto valore, che non esprime una combinazione valida, esso viene ignorato.

Come accennato il tipo di operazione eseguito da **mount** viene stabilito in base al contenuto di **mountflags**, la scelta viene effettuata controllando nell'ordine:

1. se è presente il flag **MS_REMOUNT**, nel qual caso verrà eseguito il rimontaggio del filesystem, con le nuove opzioni indicate da **data** e dagli altri flag di **mountflags**;
2. se è presente il flag **MS_BIND**, nel qual caso verrà eseguito un *bind mount* (argomento che tratteremo più avanti);
3. se è presente uno fra **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, **MS_UNBINDABLE**, nel qual caso verrà cambiata la modalità di propagazione del montaggio (detti valori sono mutualmente esclusivi).
4. se è presente **MS_MOVE**, nel qual caso verrà effettuato uno spostamento del *mount point*;
5. se nessuno dei precedenti è presente si tratta di una ordinaria operazione di montaggio di un filesystem.

Il fatto che questi valori vengano controllati in quest'ordine significa che l'effetto di alcuni di questi flag possono cambiare se usati in combinazione con gli altri che vengono prima nella

¹²benché queste siano espresse nel comando **mount** con l'opzione **-o** esse non vengono impostate nei valori di **data**, che serve solo per le opzioni specifiche di ogni filesystem.

¹³il valore era il *magic number* **0xC0ED**, si può usare la costante **MS_MGC_MSK** per ottenere la parte di **mountflags** riservata al *magic number*, mentre per specificarlo si può dare un OR aritmetico con la costante **MS_MGC_VAL**.

sequenza (è quanto avviene ad esempio per `MS_BIND` usato con `MS_REMOUNT`). Tratteremo questi *mount flags* speciali per primi, nell'ordine appena illustrato, tornando sugli altri più avanti.

Usando il flag `MS_REMOUNT` si richiede a `mount` di rimontare un filesystem già montato cambiandone le opzioni di montaggio in maniera atomica (non è cioè necessario smontare e rimontare il filesystem per effettuare il cambiamento). Questa operazione consente di modificare le opzioni del filesystem anche se questo è in uso. Gli argomenti `source` e `target` devono essere gli stessi usati per il montaggio originale, mentre sia `data` che `mountflags` conterranno le nuove opzioni, `filesystemtype` viene ignorato. Perché l'operazione abbia successo occorre comunque che il cambiamento sia possibile (ad esempio non sarà possibile rimontare in sola lettura un filesystem su cui sono aperti file per la lettura/scrittura).

Qualunque opzione specifica del filesystem indicata con `data` può essere modificata (ma si dovranno rielencare tutte quelle volute), mentre con `mountflags` possono essere modificate solo alcune opzioni generiche: `MS_LAZYTIME`, `MS_MANDLOCK`, `MS_NOATIME`, `MS_NODEV`, `MS_NODIRATIME`, `MS_NOEXEC`, `MS_NOSUID`, `MS_RELATIME`, `MS_RDONLY`, `MS_STRICTATIME` e `MS_SYNCHRONOUS`. Inoltre dal kernel 3.17 il comportamento relativo alle opzioni che operano sui tempi di ultimo accesso dei file (vedi sez. 4.3.4) è cambiato e se non si è indicato nessuno dei vari `MS_*ATIME` vengono mantenute le impostazioni esistenti anziché forzare l'uso di `MS_RELATIME`.

Usando il flag `MS_BIND` si richiede a `mount` di effettuare un cosiddetto *bind mount*, l'operazione che consente di montare una directory di un filesystem in un'altra directory. L'opzione è disponibile a partire dai kernel della serie 2.4. In questo caso verranno presi in considerazione solo gli argomenti `source`, che stavolta indicherà la directory che si vuole montare e non un file di dispositivo, e `target` che indicherà la directory su cui verrà effettuato il *bind mount*. Gli argomenti `filesystemtype` e `data` vengono ignorati.

Quello che avviene con questa operazione è che in corrispondenza del *pathname* indicato da `target` viene montato l'*inode* di `source`, così che la porzione di albero dei file presente sotto `source` diventi visibile allo stesso modo sotto `target`. Trattandosi esattamente dei dati dello stesso filesystem, ogni modifica fatta in uno qualunque dei due rami di albero sarà visibile nell'altro, visto che entrambi faranno riferimento agli stessi *inode*.

Dal punto di vista del VFS l'operazione è analoga al montaggio di un filesystem proprio nel fatto che anche in questo caso si inserisce in corrispondenza della *dentry* di `target` un diverso *inode*, che stavolta, invece di essere quello della radice del filesystem indicato da un file di dispositivo, è quello di una directory già montata.

Si tenga presente che proprio per questo sotto `target` comparirà il contenuto che è presente sotto `source` all'interno del filesystem in cui quest'ultima è contenuta. Questo potrebbe non corrispondere alla porzione di albero che sta sotto `source` qualora in una sottodirectory di quest'ultima si fosse effettuato un altro montaggio. In tal caso infatti nella porzione di albero sotto `source` si troverebbe il contenuto del nuovo filesystem (o di un altro *bind mount*) mentre sotto `target` ci sarebbe il contenuto presente nel filesystem originale.

L'unico altro *mount flag* usabile direttamente con `MS_BIND` è `MS_REC` che consente di eseguire una operazione di *bind mount* ricorsiva, in cui sotto `target` vengono montati ricorsivamente anche tutti gli eventuali ulteriori *bind mount* già presenti sotto `source`.

E' però possibile, a partire dal kernel 2.6.26, usare questo flag insieme a `MS_REMOUNT`, nel qual caso consente di effettuare una modifica delle opzioni di montaggio del *bind mount* ed in particolare effettuare il cosiddetto *read-only bind mount* in cui viene onorata anche la presenza aggiuntiva del flag `MS_RDONLY`. In questo modo si ottiene che l'accesso ai file sotto `target` sia effettuabile esclusivamente in sola lettura, mantenendo il normale accesso in lettura/scrittura sotto `source`.

Il supporto per il *bind mount* consente di superare i limiti presenti per gli *hard link* (di cui parleremo in sez. 4.2.1) con la possibilità di fare riferimento alla porzione dell'albero dei

file di un filesystem presente a partire da una certa directory utilizzando una qualunque altra directory, anche se questa sta su un filesystem diverso.¹⁴ Si può così fornire una alternativa all'uso dei collegamenti simbolici (di cui parleremo in sez. 4.2.1) che funziona correttamente anche all'intero di un *chroot* (argomento su cui torneremo in sez. 4.5.4).

I quattro flag `MS_PRIVATE`, `MS_SHARED`, `MS_SLAVE` e `MS_UNBINDABLE` sono stati introdotti a partire dal kernel 2.6.15 per realizzare l'infrastruttura dei cosiddetti *shared subtree*, che estendono le funzionalità dei *bind mount*. La funzionalità nasce dalle esigenze di poter utilizzare a pieno le funzionalità di isolamento fornite dal kernel per i processi (i *namespace*, che tratteremo in sez. 9.3.2) in particolare per quanto riguarda la possibilità di far avere ad un processo una visione ristretta dei filesystem montati (il *mount namespace*), ma l'applicazione è comunque rilevante anche con un classico *chroot* (vedi sez. 4.5.4).

Abbiamo visto come nella modalità ordinaria in cui si esegue un *bind mount* sotto *target* compaia lo stesso ramo di albero dei file presente sotto *source*, ma limitato a quanto presente nel filesystem di *source*; i risultati di un eventuale “*submount*” effettuato all'interno di *source* non saranno visibili. Ed anche se quelli presenti al momento dell'uso di `MS_BIND` possono essere riottenuti usando `MS_REC`, ogni eventuale “*submount*” successivo (che avvenga sotto *source* o sotto *target*) resterà “privato” al ramo di albero su cui è avvenuto.

Ci sono casi però in cui può risultare utile che eventuali “*submount*” siano visibili sui rami di albero presenti al di sotto di tutte le directory coinvolte in un *bind mount*, anche se effettuati in un secondo tempo. Per poter ottenere questa funzionalità i *bind mount* sono stati estesi introducendo i *mount peer group*, che consentono di raggrupparli in modo da poter inviare a ciascuno di essi tutti gli eventi relativi a montaggi o smontaggi effettuati al loro interno ed avere sempre una propagazione degli stessi che li renda coerenti.

Quando si effettua un montaggio ordinario, o si esegue un *bind mount*, di default non viene utilizzato nessun *mount peer group* ed il *mount point* viene classificato come “privato”, nel senso che abbiamo appena visto. Si può però marcare un *mount point* come “condiviso”, ed in questo modo esso verrà associato ad un *mount peer group* insieme a tutti gli altri ulteriori *mount point* per i quali sia stato eseguito un *bind mount*. Questo fa sì che tutte le volte che si effettua un montaggio o uno smontaggio all'interno di uno qualunque dei *mount point* del gruppo, questo venga propagato anche su tutti gli altri e sotto tutti sia visibile sempre lo stesso ramo di albero dei file.

A completare l'infrastruttura degli *shared subtree* sono state previste due ulteriori funzionalità: la prima è quella di marcare un *mount point* come “*slave*”, in tal caso le operazioni di montaggio e smontaggio effettuate al suo interno non verranno più propagate agli altri membri del *mount peer group* di cui fa parte, ma continuerà a ricevere quelle eseguite negli altri membri.

La seconda funzionalità è quella di marcare un *mount point* come “*unbindable*”; questo anzitutto impedirà che possa essere usato come sorgente di un *bind mount* ed inoltre lo renderà privato, con la conseguenza che quando è presente all'interno di altri *bind mount*, all'interno di questi si vedrà solo il contenuto originale e non quello risultante da eventuali ulteriori montaggi effettuati al suo interno.

I *mount flag* che controllano le operazioni relative agli *shared subtree* sono descritti nella lista seguente. Si ricordi che sono mutuamente esclusivi, e compatibili solo con l'uso degli ulteriori flag `MS_REC` (che applica ricorsivamente l'operazione a tutti gli eventuali *mount point* sottostanti) e `MS_SILENT`; in tutti gli altri casi *mount* fallirà con un errore di `EINVAL`. L'uni-

¹⁴e non c'è neanche il problema di non esser più in grado di cancellare un *hard link* ad una directory sullo stesso filesystem (vedi sez. 4.2.1), per cui su Linux questi non sono possibili, dato che in questo caso per la rimozione del collegamento basta smontare *target*.

co altro argomento che deve essere specificato quando li si usano è **target**; **source**, **data** e **filesystem** sono ignorati.

MS_PRIVATE Marca un *mount point* come *private mount*. Di default, finché non lo si marca altrimenti con una delle altre opzioni dell'interfaccia, ogni *mount point* è privato. Ogni *bind mount* ottenuto da un *mount point* privato si comporta come descritto nella trattazione di **MS_BIND**. Si usa questo flag principalmente per revocare gli effetti delle altre opzioni e riportare il comportamento a quello ordinario.

MS_SHARED Marca un *mount point* come *shared mount*. Lo scopo dell'opzione è ottenere che tutti i successivi *bind mount* ottenuti da un *mount point* così marcato siano di tipo *shared* e vengano inseriti nello stesso *mount peer group* in modo da “condividere” ogni ulteriore operazione di montaggio o smontaggio. Con questa opzione le operazioni di montaggio e smontaggio effettuate al di sotto di uno *shared mount* vengono automaticamente “propagate” a tutti gli altri membri del *mount peer group* di cui fa parte, in modo che la sezione di albero dei file visibile al di sotto di ciascuno di essi sia sempre la stessa.

MS_SLAVE Marca un *mount point* come *slave mount*. Se il *mount point* è parte di un *mount peer group* esso diventerà di tipo *slave*: le operazioni di montaggio e smontaggio al suo interno non verranno più propagate agli altri membri del gruppo, ma continuerà a ricevere quelle eseguite negli altri membri. Se non esistono altri membri nel gruppo il *mount point* diventerà privato, negli altri casi non subirà nessun cambiamento.

MS_UNBINDABLE Marca un *mount point* come *unbindable mount*. Un *mount point* marcato in questo modo non può essere usato per un *bind mount* del suo contenuto. Si comporta cioè come allo stesso modo di un *mount point* ordinario di tipo *private* con in più la restrizione che nessuna sua sottodirectory (anche se relativa ad un ulteriore montaggio) possa essere utilizzata come sorgente di un *bind mount*.

L'ultimo *mount flag* che controlla una modalità operativa di **mount** è **MS_MOVE**, che consente di effettuare lo spostamento del *mount point* di un filesystem. La directory del *mount point* originale deve essere indicata nell'argomento **source**, e la sua nuova posizione nell'argomento **target**. Tutti gli altri argomenti della funzione vengono ignorati.

Lo spostamento avviene atomicamente, ed il ramo di albero presente sotto **source** sarà immediatamente visibile sotto **target**. Non esiste cioè nessun momento in cui il filesystem non risulti montato in una o nell'altra directory e pertanto è garantito che la risoluzione di *pathname* relativi all'interno del filesystem non possa fallire.

Elenchiamo infine i restanti *mount flag*, il cui utilizzo non attiene alle operazioni di **mount**, ma soltanto l'impostazione di opzioni generiche relative al funzionamento di un filesystem e che vengono per lo più utilizzati solo in fase di montaggio:

MS_DIRSYNC Richiede che ogni modifica al contenuto di una directory venga immediatamente registrata su disco in maniera sincrona (introdotta a partire dai kernel della serie 2.6). L'opzione si applica a tutte le directory del filesystem, ma su alcuni filesystem è possibile impostarla a livello di singole directory o per i sotto-rami di una directory con il comando **chattr**.¹⁵

¹⁵questo avviene tramite delle opportune **ioctl** (vedi sez. 5.2.5).

Questo consente di ridurre al minimo il rischio di perdita dei dati delle directory in caso di crollo improvviso del sistema, al costo di una certa perdita di prestazioni dato che le funzioni di scrittura relative ad operazioni sulle directory non saranno più bufferizzate e si bloccheranno fino all'arrivo dei dati sul disco prima che un programma possa proseguire.

MS_LAZYTIME Modifica la modalità di registrazione di tempi dei file (vedi sez. 4.3.4) per ridurre al massimo gli accessi a disco (particolarmente utile per i portatili). Attivandolo i tempi dei file vengono mantenuti in memoria e vengono salvati su disco solo in quattro casi: quando c'è da eseguire un aggiornamento dei dati dell'*inode* per altri motivi, se viene usata una delle funzioni di sincronizzazione dei dati su disco (vedi sez. 5.2.3), se l'*inode* viene rimosso dalla memoria, o se è passato un giorno dall'ultima registrazione. Introdotto a partire dal kernel 4.0.

In questo modo si possono ridurre significativamente le scritture su disco mantenendo tutte le informazioni riguardo ai tempi dei file, riducendo anche l'impatto dell'uso di **MS_STRICTTIME**. Il costo da pagare è il rischio, in caso di crash del sistema, di avere dati vecchi fino a 24 ore per quel che riguarda i tempi dei file.

MS_MANDLOCK Consente l'uso del *mandatory locking* (vedi sez. 10.1.5) sui file del filesystem. Per poterlo utilizzare effettivamente però esso dovrà essere comunque attivato esplicitamente per i singoli file impostando i permessi come illustrato in sez. 10.1.5.

MS_NOATIME Viene disabilitato sul filesystem l'aggiornamento dell'*access time* (vedi sez. 4.3.4) per qualunque tipo di file. Dato che l'aggiornamento dell'*access time* è una funzionalità la cui utilità è spesso irrilevante ma comporta un costo elevato visto che una qualunque lettura comporta comunque una scrittura su disco, questa opzione consente di disabilitarla completamente. La soluzione può risultare troppo drastica dato che l'informazione viene comunque utilizzata da alcuni programmi, per cui nello sviluppo del kernel sono state introdotte altre opzioni che forniscono soluzioni più appropriate e meno radicali.

MS_NODEV Viene disabilitato sul filesystem l'accesso ai file di dispositivo eventualmente presenti su di esso. L'opzione viene usata come misura di precauzione per rendere inutile la presenza di eventuali file di dispositivo su filesystem che non dovrebbero contenerne.¹⁶

Viene utilizzata, assieme a **MS_NOEXEC** e **MS_NOSUID**, per fornire un accesso più controllato a quei filesystem di cui gli utenti hanno il controllo dei contenuti, in particolar modo quelli posti su dispositivi rimuovibili. In questo modo si evitano alla radice possibili situazioni in cui un utente malizioso inserisce su uno di questi filesystem dei file di dispositivo con permessi "opportunamente" ampliati che gli consentirebbero di accedere anche a risorse cui non dovrebbe.

MS_NODIRATIME Viene disabilitato sul filesystem l'aggiornamento dell'*access time* (vedi sez. 4.3.4) ma soltanto per le directory. Costituisce una alternativa per **MS_NOATIME**, che elimina l'informazione per le directory, che in pratica che non viene mai utilizzata, mantenendola per i file in cui invece ha un impiego, sia pur limitato.

¹⁶si ricordi che le convenzioni del *Linux Filesystem Hierarchy Standard* richiedono che questi siano mantenuti esclusivamente sotto `/dev`.

- MS_NOEXEC** Viene disabilitata sul filesystem l'esecuzione di un qualunque file eseguibile eventualmente presente su di esso. L'opzione viene usata come misura di precauzione per rendere impossibile l'uso di programmi posti su filesystem che non dovrebbero contenerne.
- Anche in questo caso viene utilizzata per fornire un accesso più controllato a quei filesystem di cui gli utenti hanno il controllo dei contenuti. Da questo punto di vista l'opzione è meno importante delle analoghe `MS_NODEV` e `MS_NOSUID` in quanto l'esecuzione di un programma creato dall'utente pone un livello di rischio nettamente inferiore, ed è in genere consentita per i file contenuti nella sua home directory.¹⁷
- MS_NOSUID** Viene disabilitato sul filesystem l'effetto dei bit dei permessi *suid* e *sgid* (vedi sez. 4.4.2) eventualmente presenti sui file in esso contenuti. L'opzione viene usata come misura di precauzione per rendere inefficace l'effetto di questi bit per filesystem in cui non ci dovrebbero essere file dotati di questi permessi.
- Di nuovo viene utilizzata, analogamente a `MS_NOEXEC` e `MS_NODEV`, per fornire un accesso più controllato a quei filesystem di cui gli utenti hanno il controllo dei contenuti. In questo caso si evita che un utente malizioso possa inserire su uno di questi filesystem un eseguibile con il bit *suid* attivo e di proprietà dell'amministratore o di un altro utente, che gli consentirebbe di eseguirlo per conto di quest'ultimo.
- MS_RDONLY** Esegue il montaggio del filesystem in sola lettura, non sarà possibile nessuna modifica ai suoi contenuti. Viene usato tutte le volte che si deve accedere ai contenuti di un filesystem con la certezza che questo non venga modificato (ad esempio per ispezionare un filesystem corrotto). All'avvio di default il kernel monta la radice in questa modalità. Si tenga presente che se non viene indicato il filesystem verrà montato, o rimontato nel caso lo si usi con `MS_REMOUNT`, in lettura/scrittura; questo significa in sostanza che non esiste una opzione separata per indicare il montaggio in lettura/scrittura.
- MS_REC** Applica ricorsivamente a tutti i *mount point* presenti al di sotto del *mount point* indicato gli effetti della opzione degli *shared subtree* associata. In questo caso l'argomento **target** deve fare riferimento ad un *mount point* e tutti gli altri argomenti sono ignorati, ed il flag deve essere indicato con uno fra `MS_PRIVATE`, `MS_SHARED`, `MS_SLAVE` e `MS_UNBINDABLE`. Può anche essere usato con `MS_BIND` per richiedere il montaggio ricorsivo anche degli eventuali ulteriori *bind mount* presenti sotto **target**.
- MS_RELATIME** Indica di effettuare l'aggiornamento dell'*access time* sul filesystem soltanto quando questo risulti antecedente il valore corrente del *modification time* o del *change time* (per i tempi dei file si veda sez. 4.3.4). L'opzione è disponibile a partire dal kernel 2.6.20, mentre dal 2.6.30 questo è diventato il comportamento di default del sistema, che può essere riportato a quello tradizionale con l'uso di `MS_STRICTATIME`. Sempre dal 2.6.30 il comportamento è stato anche modificato e l'*access time* viene comunque aggiornato se è più vecchio di un giorno.

¹⁷cosa che renderebbe superfluo l'attivazione di questa opzione, il cui uso ha senso solo per ambienti molto controllati in cui si vuole che gli utenti eseguano solo i programmi forniti dall'amministratore.

L'opzione consente di evitare i problemi di prestazioni relativi all'aggiornamento dell'*access time* senza avere impatti negativi riguardo le funzionalità, il comportamento adottato infatti consente di rendere evidente che vi è stato un accesso dopo la scrittura, ed evitando al contempo ulteriori operazioni su disco negli accessi successivi. In questo modo l'informazione relativa al fatto che un file sia stato letto resta disponibile, ed i programmi che ne fanno uso continuano a funzionare. Con l'introduzione di questo comportamento l'uso delle alternative `MS_NOATIME` e `MS_NODIRATIME` è sostanzialmente inutile.

MS_SILENT Richiede la soppressione di alcuni messaggi di avvertimento nei log del kernel (vedi sez. 8.1.5). L'opzione è presente a partire dal kernel 2.6.17 e sostituisce, utilizzando un nome non fuorviante, la precedente `MS_VERBOSE`, introdotta nel kernel 2.6.12, che aveva lo stesso effetto.

MS_STRICTATIME Ripristina il comportamento tradizionale per cui l'*access time* viene aggiornato ad ogni accesso al file. L'opzione è disponibile solo a partire dal kernel 2.6.30 quando il comportamento di default del kernel è diventato quello fornito da `MS_RELATIME`.

MS_SYNCHRONOUS Abilita la scrittura sincrona richiedendo che ogni modifica al contenuto del filesystem venga immediatamente registrata su disco. Lo stesso comportamento può essere ottenuto con il flag `O_SYNC` di `open` (vedi sez. 5.1.2).

Questa opzione consente di ridurre al minimo il rischio di perdita dei dati in caso di crollo improvviso del sistema, al costo di una pesante perdita di prestazioni dato che tutte le funzioni di scrittura non saranno più bufferizzate e si bloccheranno fino all'arrivo dei dati sul disco. Per un compromesso in cui questo comportamento avviene solo per le directory, ed ha quindi una incidenza nettamente minore, si può usare `MS_DIRSYNC`.

Una volta che non si voglia più utilizzare un certo filesystem è possibile “smontarlo” usando la funzione di sistema `umount`, il cui prototipo è:

```
#include <sys/mount.h>
umount(const char *target)
```

Smonta un filesystem.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBUSY` il filesystem è occupato.

`EINVAL` `target` non è un *mount point*.

`EPERM` il processo non ha i privilegi di amministratore.¹⁸

ed inoltre `EFAULT`, `ELOOP`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM` nel loro significato generico.

La funzione prende il nome della directory su cui il filesystem è montato e non il file o il dispositivo che è stato montato,¹⁹ in quanto a partire dai kernel della serie 2.4.x è possibile montare lo stesso dispositivo in più punti. Nel caso più di un filesystem sia stato montato sullo stesso *mount point* viene smontato quello che è stato montato per ultimo. Si tenga presente che la funzione fallisce se il filesystem è “occupato”, cioè quando ci sono ancora dei file aperti sul

¹⁸più precisamente la capacità `CAP_SYS_ADMIN`, vedi sez. 9.1.1.

¹⁹questo è vero a partire dal kernel 2.3.99-pre7, prima esistevano due chiamate separate e la funzione poteva essere usata anche specificando il file di dispositivo.

filesystem, se questo contiene la directory di lavoro (vedi sez. 4.2.4) di un qualunque processo o il *mount point* di un altro filesystem.

Linux provvede inoltre una seconda funzione di sistema, `umount2`, che consente un maggior controllo delle operazioni, come forzare lo smontaggio di un filesystem anche quando questo risulti occupato; il suo prototipo è:

<pre>#include <sys/mount.h> umount2(const char *target, int flags)</pre>	
	Smonta un filesystem.
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EAGAIN si è chiamata la funzione con <code>MNT_EXPIRE</code> ed il filesystem non era occupato.</p> <p>EBUSY <code>target</code> è la directory di lavoro di qualche processo, o contiene dei file aperti, o un altro <i>mount point</i>.</p> <p>EINVAL <code>target</code> non è un <i>mount point</i> o si è usato <code>MNT_EXPIRE</code> con <code>MNT_FORCE</code> o <code>MNT_DETACH</code> o si è specificato un flag non esistente.</p> <p>e tutti gli altri valori visti per <code>umount</code> con lo stesso significato.</p>	

Il valore di `flags` è una maschera binaria dei flag che controllano le modalità di smontaggio, che deve essere specificato con un OR aritmetico delle costanti illustrate in tab. 4.3. Specificando `MNT_FORCE` la funzione cercherà di liberare il filesystem anche se è occupato per via di una delle condizioni descritte in precedenza. A seconda del tipo di filesystem alcune (o tutte) possono essere superate, evitando l'errore di `EBUSY`. In tutti i casi prima dello smontaggio viene eseguita una sincronizzazione dei dati.

Costante	Descrizione
<code>MNT_FORCE</code>	Forza lo smontaggio del filesystem anche se questo è occupato (presente dai kernel della serie 2.2).
<code>MNT_DETACH</code>	Esegue uno smontaggio “pigro”, in cui si blocca l'accesso ma si aspetta che il filesystem si liberi (presente dal kernel 2.4.11 e dalla <i>glibc</i> 2.11).
<code>MNT_EXPIRE</code>	Se non occupato marca un <i>mount point</i> come “in scadenza” in modo che ad una successiva chiamata senza utilizzo del filesystem questo venga smontato (presente dal kernel 2.6.8 e dalla <i>glibc</i> 2.11).
<code>UMOUNT_NOFOLLOW</code>	Non dereferenzia <code>target</code> se questo è un collegamento simbolico (vedi sez. 4.2.1) evitando problemi di sicurezza (presente dal kernel 2.6.34).

Tabella 4.3: Costanti che identificano i bit dell'argomento `flags` della funzione `umount2`.

Con l'opzione `MNT_DETACH` si richiede invece uno smontaggio “pigro” (o *lazy umount*) in cui il filesystem diventa inaccessibile per i nuovi processi subito dopo la chiamata della funzione, ma resta accessibile per quelli che lo hanno ancora in uso e non viene smontato fintanto che resta occupato.

Con `MNT_EXPIRE`, che non può essere specificato insieme agli altri due, si marca il *mount point* di un filesystem non occupato come “in scadenza”, in tal caso `umount2` ritorna con un errore di `EAGAIN`, mentre in caso di filesystem occupato si sarebbe ricevuto `EBUSY`. Una volta marcato, se nel frattempo non viene fatto nessun uso del filesystem, ad una successiva chiamata con `MNT_EXPIRE` questo verrà smontato. Questo flag consente di realizzare un meccanismo che smonti automaticamente i filesystem che restano inutilizzati per un certo periodo di tempo.

Infine il flag `UMOUNT_NOFOLLOW` non dereferenzia `target` se questo è un collegamento simbolico (vedi sez. 4.2.1). Questa è una misura di sicurezza introdotta per evitare, per quei filesystem per il quale è prevista una gestione diretta da parte degli utenti, come quelli basati

su FUSE,²⁰ che si possano passare ai programmi che effettuano lo smontaggio dei filesystem, che in genere sono privilegiati ma consentono di agire solo sui propri *mount point*, dei collegamenti simbolici che puntano ad altri *mount point*, ottenendo così la possibilità di smontare qualunque filesystem.

Altre due funzioni di sistema specifiche di Linux,²¹ utili per ottenere in maniera diretta informazioni riguardo al filesystem su cui si trova un certo file, sono `statfs` e `fstatfs`, i cui prototipi sono:

```
#include <sys/vfs.h>
int statfs(const char *path, struct statfs *buf)
int fstatfs(int fd, struct statfs *buf)
```

Restituiscono informazioni relative ad un filesystem.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

ENOSYS il filesystem su cui si trova il file specificato non supporta la funzione.

ed inoltre EFAULT ed EIO per entrambe, EBADF per `fstatfs`, ENOTDIR, ENAMETOOLONG, ENOENT, EACCES, ELOOP per `statfs` nel loro significato generico.

Queste funzioni permettono di ottenere una serie di informazioni generali riguardo al filesystem su cui si trova il file specificato con un *pathname* per `statfs` e con un file descriptor (vedi sez. 5.1.1) per `fstatfs`. Le informazioni vengono restituite all'indirizzo `buf` di una struttura `statfs` definita come in fig. 4.8, ed i campi che sono indefiniti per il filesystem in esame sono impostati a zero. I valori del campo `f_type` sono definiti per i vari filesystem nei relativi file di header dei sorgenti del kernel da costanti del tipo `XXX_SUPER_MAGIC`, dove `XXX` in genere è il nome del filesystem stesso.

```
struct statfs {
    long    f_type;        /* tipo di filesystem */
    long    f_bsize;       /* dimensione ottimale dei blocchi di I/O */
    long    f_blocks;      /* blocchi totali nel filesystem */
    long    f_bfree;       /* blocchi liberi nel filesystem */
    long    f_bavail;      /* blocchi liberi agli utenti normali */
    long    f_files;       /* inode totali nel filesystem */
    long    f_ffree;       /* inode liberi nel filesystem */
    fsid_t  f_fsid;       /* filesystem id */
    long    f_namelen;     /* lunghezza massima dei nomi dei file */
    long    f_spare[6];    /* riservati per uso futuro */
};
```

Figura 4.8: La struttura `statfs`.

La *glibc* provvede infine una serie di funzioni per la gestione dei due file `/etc/fstab`²² ed `/etc/mstab`²³ che convenzionalmente sono usati in quasi tutti i sistemi unix-like per mantenere rispettivamente le informazioni riguardo ai filesystem da montare e a quelli correntemente montati. Le funzioni servono a leggere il contenuto di questi file in opportune strutture `fstab` e `mntent`, e, nel caso di `/etc/mstab`, per inserire e rimuovere le voci presenti nel file.

²⁰il *Filesystem in USEr space* (FUSE) è una delle più interessanti applicazioni del VFS che consente, tramite un opportuno modulo, di implementarne le funzioni in *user space*, così da rendere possibile l'implementazione di un qualunque filesystem (con applicazioni di grande interesse come il filesystem cifrato *encfs* o il filesystem di rete *sshfs*) che possa essere usato direttamente per conto degli utenti.

²¹esse si trovano anche su BSD, ma con una struttura diversa.

²²più precisamente `setfsent`, `getfsent`, `getfsfile`, `getfsspec`, `endfsent`.

²³più precisamente `setmntent`, `getmntent`, `getmntent_r`, `addmntent`, `endmntent`, `hasmntopt`.

In generale si dovrebbero usare queste funzioni, in particolare quelle relative a `/etc/mtab`, quando si debba scrivere un programma che effettua il montaggio di un filesystem. In realtà in questi casi è molto più semplice invocare direttamente il programma `mount`. Inoltre l'uso stesso di `/etc/mtab` è considerato una pratica obsoleta, in quanto se non aggiornato correttamente (cosa che è impossibile se la radice è montata in sola lettura) il suo contenuto diventa fuorviante.

Per questo motivo il suo utilizzo viene deprecato ed in molti casi viene già oggi sostituito da un collegamento simbolico a `/proc/mounts`, che contiene una versione degli stessi contenuti (vale a dire l'elenco dei filesystem montati) generata direttamente dal kernel, e quindi sempre disponibile e sempre aggiornata. Per questo motivo tralasceremo la trattazione, di queste funzioni, rimandando al manuale della *glibc* [?] per la documentazione completa.

4.2 La gestione di file e directory

In questa sezione esamineremo le funzioni usate per la manipolazione dei nomi file e directory, per la creazione di collegamenti simbolici e diretti, per la gestione e la lettura delle directory. In particolare ci soffermeremo sulle conseguenze che derivano dalla architettura di un filesystem unix-like illustrata in sez. 4.1.2 per quanto attiene il comportamento e gli effetti delle varie funzioni. Tratteremo infine la directory di lavoro e le funzioni per la gestione di file speciali e temporanei.

4.2.1 La gestione dei nomi dei file

Una caratteristica comune a diversi sistemi operativi è quella di poter creare dei nomi alternativi, come gli alias del vecchio MacOS o i collegamenti di Windows o i nomi logici del VMS, che permettono di fare riferimento allo stesso file chiamandolo con nomi diversi o accedendovi da directory diverse. Questo è possibile anche in ambiente Unix, dove un nome alternativo viene usualmente chiamato “collegamento” (o *link*). Data l'architettura del sistema riguardo la gestione dei file vedremo però che ci sono due metodi sostanzialmente diversi per fare questa operazione.

In sez. 4.1.2 abbiamo spiegato come la capacità di chiamare un file con nomi diversi sia connaturata con l'architettura di un filesystem per un sistema Unix, in quanto il nome del file che si trova in una directory è solo un'etichetta associata ad un puntatore che permette di ottenere il riferimento ad un *inode*, e che è quest'ultimo che viene usato dal kernel per identificare univocamente gli oggetti sul filesystem.

Questo significa che fintanto che si resta sullo stesso filesystem la realizzazione di un *link* è immediata: uno stesso file può avere tanti nomi diversi, dati da altrettante associazioni diverse allo stesso *inode* effettuate tramite “etichette” diverse in directory diverse. Si noti anche come nessuno di questi nomi possa assumere una particolare preferenza o originalità rispetto agli altri, in quanto tutti fanno comunque riferimento allo stesso *inode* e quindi tutti otterranno lo stesso file.

Quando si vuole aggiungere ad una directory una voce che faccia riferimento ad un file già esistente come appena descritto, per ottenere quello che viene denominato “collegamento diretto” (o *hard link*), si deve usare la funzione di sistema `link`, il cui prototipo è:

<pre>#include <unistd.h> int link(const char *oldpath, const char *newpath)</pre>	
	Crea un nuovo collegamento diretto (<i>hard link</i>).
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EEXIST un file (o una directory) di nome <code>newpath</code> esiste già.</p> <p>EMLINK ci sono troppi collegamenti al file <code>oldpath</code> (il numero massimo è specificato dalla variabile <code>LINK_MAX</code>, vedi sez. 6.1.1).</p> <p>EPERM il filesystem che contiene <code>oldpath</code> e <code>newpath</code> non supporta i collegamenti diretti, è una directory o per <code>oldpath</code> non si rispettano i criteri per i <i>protected hardlink</i>.²⁴</p> <p>EXDEV i file <code>oldpath</code> e <code>newpath</code> non fanno riferimento ad un filesystem montato sullo stesso <i>mount point</i>.</p> <p>ed inoltre EACCES, EDQUOT, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EROFS nel loro significato generico.</p>	

La funzione crea in `newpath` un collegamento diretto al file indicato da `oldpath`. Per quanto detto la creazione di un nuovo collegamento diretto non copia il contenuto del file, ma si limita a creare la voce specificata da `newpath` nella directory corrispondente e l'unica proprietà del file che verrà modificata sarà il numero di riferimenti al file (il campo `i_nlink` della struttura `inode`, vedi fig. 4.2) che verrà aumentato di uno. In questo modo lo stesso file potrà essere acceduto sia con `newpath` che con `oldpath`.

Per quanto dicevamo in sez. 4.1.2 la creazione di un collegamento diretto è possibile solo se entrambi i *pathname* sono nello stesso filesystem ed inoltre esso deve supportare gli *hard link* (il meccanismo non è disponibile ad esempio con il filesystem *vfat* di Windows). In realtà la funzione ha un ulteriore requisito, e cioè che non solo che i due file siano sullo stesso filesystem, ma anche che si faccia riferimento ad essi all'interno dello stesso *mount point*.²⁵ La funzione inoltre opera sia sui file ordinari che sugli altri oggetti del filesystem, con l'eccezione delle directory. In alcune versioni di Unix solo l'amministratore è in grado di creare un collegamento diretto ad un'altra directory: questo viene fatto perché con una tale operazione è possibile creare dei *loop* nel filesystem (vedi fig. 4.9) la cui rimozione diventerebbe piuttosto complicata.²⁶

Data la pericolosità di questa operazione, e visto che i collegamenti simbolici (che tratteremo a breve) ed i *bind mount* (già visti in sez. 4.1.4) possono fornire la stessa funzionalità senza questi problemi, nel caso di Linux questa capacità è stata completamente disabilitata, e al tentativo di creare un collegamento diretto ad una directory la funzione `link` restituisce sempre l'errore `EPERM`.

Un ulteriore comportamento peculiare di Linux è quello in cui si crea un *hard link* ad un collegamento simbolico. In questo caso lo standard POSIX.1-2001 prevederebbe che quest'ultimo venga risolto e che il collegamento sia effettuato rispetto al file cui esso punta, e che venga riportato un errore qualora questo non esista o non sia un file. Questo era anche il comportamento iniziale di Linux ma a partire dai kernel della serie 2.0.x²⁷ è stato adottato un comportamento che non segue più lo standard per cui l'*hard link* viene creato nei confronti

²⁴i *protected hardlink* sono una funzionalità di protezione introdotta con il kernel 3.16 (si veda sez. 9.1.3 per i dettagli) che limita la capacità di creare un *hard link* ad un file qualunque.

²⁵si tenga presente infatti, come detto in sez. 4.1.4, che a partire dal kernel 2.4 uno stesso filesystem può essere montato più volte su directory diverse.

²⁶occorrerebbe infatti eseguire il programma `fsck` per riparare il filesystem, perché in caso di *loop* la directory non potrebbe essere più svuotata, contenendo comunque se stessa, e quindi non potrebbe essere rimossa.

²⁷per la precisione il comportamento era quello previsto dallo standard POSIX fino al kernel di sviluppo 1.3.56, ed è stato temporaneamente ripristinato anche durante lo sviluppo della serie 2.1.x, per poi tornare al comportamento attuale fino ad oggi (per riferimento si veda <http://lwn.net/Articles/293902>).

del collegamento simbolico, e non del file cui questo punta. La revisione POSIX.1-2008 lascia invece il comportamento dipendente dall'implementazione, cosa che rende Linux conforme a questa versione successiva dello standard.

La ragione di questa differenza rispetto al vecchio standard, presente anche in altri sistemi unix-like, è dovuta al fatto che un collegamento simbolico può fare riferimento anche ad un file non esistente o a una directory, per i quali l'*hard link* non può essere creato, per cui la scelta di seguire il collegamento simbolico è stata ritenuta una scelta scorretta nella progettazione dell'interfaccia. Infatti se non ci fosse il comportamento adottato da Linux sarebbe impossibile creare un *hard link* ad un collegamento simbolico, perché la funzione lo risolverebbe e l'*hard link* verrebbe creato verso la destinazione. Invece evitando di seguire lo standard l'operazione diventa possibile, ed anche il comportamento della funzione risulta molto più comprensibile. Tanto più che se proprio se si vuole creare un *hard link* rispetto alla destinazione di un collegamento simbolico è sempre possibile farlo direttamente.²⁸

Dato che *link* crea semplicemente dei nomi che fanno riferimenti agli *inode*, essa può funzionare soltanto per file che risiedono sullo stesso filesystem e solo per un filesystem di tipo Unix. Inoltre abbiamo visto che in Linux non è consentito eseguire un collegamento diretto ad una directory.

Per ovviare a queste limitazioni, come accennato all'inizio, i sistemi unix-like supportano un'altra forma di collegamento, detta "*collegamento simbolico*" (o anche *soft link* o *symbolic link*). In questo caso si tratta, come avviene in altri sistemi operativi, di file speciali che contengono semplicemente il riferimento ad un altro file (o directory). In questo modo è possibile effettuare *link* anche attraverso filesystem diversi, a file posti in filesystem che non supportano i collegamenti diretti, a delle directory, ed anche a file che non esistono ancora.

Il meccanismo funziona in quanto i *symbolic link* sono riconosciuti come tali dal kernel²⁹ e tutta una serie di funzioni di sistema (come *open* o *stat*) quando ricevono come argomento il *pathname* di un collegamento simbolico vanno automaticamente ad operare sul file da esso specificato. La funzione di sistema che permette di creare un nuovo collegamento simbolico è *symlink*, ed il suo prototipo è:

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath)
    Crea un nuovo collegamento simbolico (symbolic link).
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EACCES o non si hanno i permessi sulla directory in cui creare il *link*.

EEXIST esiste già un file *newpath*.

ENOENT una componente di *newpath* non esiste o *oldpath* è una stringa vuota.

EPERM il filesystem che contiene *newpath* non supporta i collegamenti simbolici.

EROFS *newpath* è su un filesystem montato in sola lettura.

ed inoltre EDQUOT, EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOMEM, ENOSPC e ENOTDIR nel loro significato generico.

La funzione crea un nuovo collegamento simbolico *newpath* che fa riferimento ad *oldpath*. Si tenga presente che la funzione non effettua nessun controllo sull'esistenza di un file di nome *oldpath*, ma si limita ad inserire il *pathname* nel collegamento simbolico. Pertanto un collegamento simbolico può anche riferirsi ad un file che non esiste ed in questo caso si ha

²⁸ciò non toglie che questo comportamento possa causare problemi, come nell'esempio descritto nell'articolo citato nella nota precedente, a programmi che non si aspettano questa differenza rispetto allo standard POSIX.

²⁹è uno dei diversi tipi di file visti in tab. 1.1, contrassegnato come tale nell'*inode* e riconoscibile dal valore del campo *st_mode* della struttura *stat* (vedi sez. 4.3.1).

quello che viene chiamato un *dangling link*, letteralmente un “collegamento ciondolante”. Ad esempio possiamo usare il comando `ln` per creare un collegamento simbolico nella nostra directory con:

```
piccardi@hain:~/gapil$ ln -s /tmp/tmp_file symlink
```

e questo avrà successo anche se `/tmp/tmp_file` non esiste:

```
piccardi@hain:~/gapil$ ls symlink
symlink
```

ma questo può generare confusione, perché accedendo in lettura a `symlink`, ad esempio con `cat`, otterremmo:

```
piccardi@hain:~/gapil$ cat symlink
cat: symlink: No such file or directory
```

con un errore che può sembrare sbagliato, dato che `ls` ci ha mostrato in precedenza l'esistenza di `symlink`. Se invece andassimo a scrivere su `symlink`, l'effetto sarebbe quello di ottenere la creazione di `/tmp/tmp_file` (che a quel punto verrebbe creato) senza errori.

Come accennato i collegamenti simbolici sono risolti automaticamente dal kernel all'invocazione delle varie *system call*. In tab. 4.4 si è riportato un elenco dei comportamenti delle varie funzioni di sistema che operano sui file nei confronti della risoluzione dei collegamenti simbolici, specificando quali li seguono e quali invece possono operare direttamente sui loro contenuti.

Funzione	Segue il link	Non segue il link
access	•	—
chdir	•	—
chmod	•	—
chown	—	•
creat	•	—
exec	•	—
lchown	•	—
link ³⁰	—	•
lstat	—	•
mkdir	•	—
mkfifo	•	—
mknod	•	—
open	•	—
opendir	•	—
pathconf	•	—
readlink	—	•
remove	—	•
rename	—	•
stat	•	—
truncate	•	—
unlink	—	•

Tabella 4.4: Uso dei collegamenti simbolici da parte di alcune funzioni.

Si noti che non si è specificato il comportamento delle funzioni che operano con i file descriptor (che tratteremo nel prossimo capitolo), in quanto la risoluzione del collegamento simbolico viene in genere effettuata dalla funzione che restituisce il file descriptor (normalmente la `open`, vedi sez. 5.1.2) e tutte le operazioni seguenti fanno riferimento solo a quest'ultimo.

³⁰a partire dalla serie 2.0, e contrariamente a quanto indicato dallo standard POSIX.1-2001.

Si tenga anche presente che a partire dal kernel 3.16, se si abilita la funzionalità dei *protected symlinks* (attiva di default in tutte le distribuzioni più recenti) la risoluzione dei nomi attraverso un collegamento simbolico può fallire per una serie di restrizioni di sicurezza aggiuntive imposte dal meccanismo (si consulti sez. 9.1.3 per i dettagli).

Dato che, come indicato in tab. 4.4, funzioni come la `open` seguono i collegamenti simbolici, occorrono funzioni apposite per accedere alle informazioni del collegamento invece che a quelle del file a cui esso fa riferimento. Quando si vuole leggere il contenuto di un collegamento simbolico si usa la funzione di sistema `readlink`, il cui prototipo è:

<pre>#include <unistd.h> int readlink(const char *pathname, char *buff, size_t size)</pre> <p style="text-align: right;">Legge il contenuto di un collegamento simbolico.</p>
<p>La funzione ritorna il numero di caratteri letti dentro <code>buff</code> in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EACCES</code> non si hanno i permessi di attraversamento di una delle directory del <code>pathname</code></p> <p><code>EINVAL</code> <code>pathname</code> non è un collegamento simbolico o <code>size</code> non è positiva.</p> <p>ed inoltre <code>EFAULT</code>, <code>EIO</code>, <code>ELOOP</code>, <code>ENAMETOOLONG</code>, <code>ENOENT</code>, <code>ENOMEM</code> e <code>ENOTDIR</code> nel loro significato generico.</p>

La funzione legge il *pathname* a cui fa riferimento il collegamento simbolico indicato dall'argomento `pathname` scrivendolo sul buffer `buff` di dimensione `size`. Si tenga presente che la funzione non termina la stringa con un carattere nullo e che se questa è troppo lunga la tronca alla dimensione specificata da `size` per evitare di scrivere dati oltre le dimensioni del buffer.

Figura 4.9: Esempio di loop nel filesystem creato con un collegamento simbolico.

Come accennato uno dei motivi per cui non sono consentiti *hard link* alle directory è che questi possono creare dei *loop* nella risoluzione dei nomi che non possono essere eliminati facilmente. Invece è sempre possibile, ed in genere anche molto utile, creare un collegamento simbolico ad una directory, anche se in questo caso si potranno ottenere anche dei *loop*.

La situazione è illustrata in fig. 4.9, che riporta la struttura della directory `/boot`. Come si vede si è creato al suo interno un collegamento simbolico che punta di nuovo a `/boot`.³¹ Un *loop* di di questo tipo però può causare problemi per tutti i programmi che effettuano la scansione di una directory, e ad esempio se lanciassimo un comando come `grep -r linux *`, il *loop* nella directory porterebbe ad esaminare `/boot`, `/boot/boot`, `/boot/boot/boot` e così via.

Per questo motivo il kernel e le librerie prevedono che nella risoluzione di un *pathname* possano essere seguiti fino ad un certo numero massimo di collegamenti simbolici, il cui valore limite è specificato dalla costante `MAXSYMLINKS`. Se il limite viene superato si ha un errore ed `errno` viene impostata al valore `ELOOP`, che nella quasi totalità dei casi indica appunto che si è creato un collegamento simbolico che fa riferimento ad una directory del suo stesso *pathname*.

Un'altra funzione relativa alla gestione dei nomi dei file, anche se a prima vista parrebbe riguardare un argomento completamente diverso, è quella per la cancellazione di un file. In realtà una *system call* che serva proprio a cancellare un file non esiste neanche perché, come accennato in sez. 4.1.2, quando in un sistema unix-like si richiede la rimozione di un file, quello che si va a cancellare è soltanto la voce che referencia il suo *inode* all'interno di una directory.

³¹il *loop* mostrato in fig. 4.9 è stato usato per poter permettere a al *bootloader* `grub` di vedere i file contenuti nella directory `/boot` con lo stesso *pathname* con cui verrebbero visti dal sistema operativo, anche quando si trovano, come accade spesso, su una partizione separata (che `grub` all'avvio vedrebbe come `/`).

La funzione di sistema che consente di effettuare questa operazione, il cui nome come si può notare ha poco a che fare con il concetto di rimozione, è `unlink`, ed il suo prototipo è:

<pre>#include <unistd.h> int unlink(const char *pathname)</pre>		Cancella un file.
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori: ³²		
EACCES	non si ha il permesso di scrittura sulla directory che contiene <code>pathname</code> o quello di attraversamento per una delle directory superiori.	
EBUSY	<code>pathname</code> non può essere rimosso perché è in uso da parte del sistema (in particolare per i cosiddetti <i>silly renames</i> di NFS).	
EISDIR	<code>pathname</code> si riferisce ad una directory.	
EPERM	il filesystem non consente l'operazione, o la directory che contiene <code>pathname</code> ha lo <i>sticky bit</i> e non si è il proprietario del file o non si hanno privilegi amministrativi.	
ed inoltre EFAULT, EIO, ELOOP, ENOENT, ENOMEM, ENOTDIR, EROFS nel loro significato generico.		

La funzione elimina il nome specificato dall'argomento `pathname` nella directory che lo contiene e decrementa il numero di riferimenti nel relativo *inode*; come per `link` queste due operazioni sono effettuate all'interno della *system call* in maniera atomica rispetto ai processi.

Si ricordi che, anche se se ne è rimosso il nome, un file viene realmente cancellato soltanto quando il numero di collegamenti mantenuto nell'*inode* diventa nullo; solo allora l'*inode* viene disallocato e lo spazio che il file occupava sul disco viene liberato.

Si tenga presente comunque che a questo si aggiunge sempre un'ulteriore condizione e cioè che non ci siano processi che stiano ancora lavorando sul file. Come vedremo in sez. 5.1 il kernel una tabella di tutti file aperti da ciascun processo, che a sua volta contiene i riferimenti agli *inode* ad essi relativi. Prima di procedere alla cancellazione dello spazio occupato su disco dal contenuto di un file il kernel controlla anche questa tabella, per verificare che anche in essa non ci sia più nessun riferimento all'*inode* in questione, assicurandosi con questo che nessun processo stia ancora usando il file.

Nel caso di socket, *fifo* o file di dispositivo la funzione rimuove il nome, e come per i file normali i processi che hanno aperto uno di questi oggetti possono continuare ad utilizzarli. Nel caso di cancellazione di un *link* simbolico, che consiste solo nel rimando ad un altro file, questo viene immediatamente eliminato e non sarà più utilizzabile.

Per cancellare una voce in una directory è necessario avere il permesso di scrittura su di essa, dato che si va a rimuovere una voce dal suo contenuto, e il diritto di esecuzione/attraversamento sulla directory che la contiene (affronteremo in dettaglio l'argomento dei permessi di file e directory in sez. 4.4). Se inoltre per la directory è impostato lo *sticky bit* (vedi sez. 4.4.2), occorrerà anche essere proprietari del file o proprietari della directory o avere i privilegi di amministratore.

Questa caratteristica del sistema, che consente di usare un file anche se lo si è "cancellato", può essere usata per essere sicuri di non lasciare file temporanei su disco in caso di uscita imprevista di un programma. La tecnica è quella di aprire un nuovo file e chiamare `unlink` su di esso subito dopo, in questo modo il contenuto del file sarà sempre disponibile all'interno del processo attraverso il suo file descriptor (vedi sez. 5.1.1), ma non ne resterà traccia in nessuna

³²questa funzione su Linux ha alcune peculiarità nei codici di errore, in particolare riguardo la rimozione delle directory che non è mai permessa e che causa l'errore EISDIR; questo è un valore specifico di Linux non conforme allo standard POSIX che prescrive invece l'uso di EPERM in caso l'operazione non sia consentita o il processo non abbia privilegi sufficienti, valore che invece Linux usa anche se il filesystem non supporta la funzione, inoltre il codice EBUSY nel caso la directory sia occupata su Linux non esiste.

directory, inoltre lo spazio occupato su disco verrà immediatamente rilasciato alla conclusione del processo, quando tutti i file vengono chiusi.

Al contrario di quanto avviene con altri Unix, in Linux non è possibile usare la funzione `unlink` sulle directory, che in tal caso fallisce con un errore di `EISDIR`. Per cancellare una directory si deve usare la apposita funzione di sistema `rmdir` (che vedremo in sez. 4.2.2), oppure la funzione `remove`.

Quest'ultima è la funzione prevista dallo standard ANSI C per effettuare una cancellazione generica di un file o di una directory e viene usata in generale anche per i sistemi operativi che non supportano gli *hard link*. Nei sistemi unix-like `remove` è equivalente ad usare in maniera trasparente `unlink` per i file ed `rmdir` per le directory; il suo prototipo è:

<pre>#include <stdio.h> int remove(const char *pathname)</pre>	Cancella un file o una directory.
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori relativi alla chiamata utilizzata, pertanto si può fare riferimento a quanto illustrato nelle descrizioni di <code>unlink</code> e <code>rmdir</code> .	

La funzione utilizza la funzione `unlink` per cancellare i file (e si applica anche a link simbolici, socket, *fifo* e file di dispositivo) e la funzione `rmdir` (vedi sez. 4.2.2) per cancellare le directory.³³ Si tenga presente che, per alcune limitazioni del protocollo NFS, utilizzare questa funzione su file che usano questo filesystem di rete può comportare la scomparsa di file ancora in uso.

Infine per cambiare nome ad un file o a una directory si usa la funzione di sistema `rename`,³⁴ il cui prototipo è:

<pre>#include <stdio.h> int rename(const char *oldpath, const char *newpath)</pre>	Rinomina un file o una directory.																
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:																	
<table> <tr> <td>EACCESS</td> <td>manca il permesso di scrittura sulle directory contenenti <code>oldpath</code> e <code>newpath</code> o di attraversare il loro <code>pathname</code> o di scrivere su <code>newpath</code> se questa è una directory.</td> </tr> <tr> <td>EBUSY</td> <td><code>oldpath</code> o <code>newpath</code> sono in uso da parte di qualche processo (come directory di lavoro o come radice) o del sistema (come <i>mount point</i>) ed il sistema non riesce a risolvere la situazione.</td> </tr> <tr> <td>EEXIST</td> <td><code>newpath</code> è una directory già esistente e non è vuota (anche <code>ENOTEMPTY</code>).</td> </tr> <tr> <td>EINVAL</td> <td><code>newpath</code> contiene un prefisso di <code>oldpath</code> o più in generale si è cercato di creare una directory come sotto-directory di sé stessa.</td> </tr> <tr> <td>EISDIR</td> <td><code>newpath</code> è una directory mentre <code>oldpath</code> non è una directory.</td> </tr> <tr> <td>ENOTDIR</td> <td>uno dei componenti dei <code>pathname</code> non è una directory o <code>oldpath</code> è una directory e <code>newpath</code> esiste e non è una directory.</td> </tr> <tr> <td>EPERM</td> <td>la directory contenente <code>oldpath</code> o quella contenente un <code>newpath</code> esistente hanno lo <i>sticky bit</i> e non si è i proprietari dei rispettivi file (o non si hanno privilegi amministrativi) oppure il filesystem non supporta l'operazione.</td> </tr> <tr> <td>EXDEV</td> <td><code>oldpath</code> e <code>newpath</code> non sono sullo stesso filesystem e sotto lo stesso <i>mount point</i>.</td> </tr> </table> ed inoltre <code>EFAULT</code> , <code>ELOOP</code> , <code>EMLINK</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> , <code>ENOMEM</code> , <code>ENOSPC</code> e <code>EROFS</code> nel loro significato generico.		EACCESS	manca il permesso di scrittura sulle directory contenenti <code>oldpath</code> e <code>newpath</code> o di attraversare il loro <code>pathname</code> o di scrivere su <code>newpath</code> se questa è una directory.	EBUSY	<code>oldpath</code> o <code>newpath</code> sono in uso da parte di qualche processo (come directory di lavoro o come radice) o del sistema (come <i>mount point</i>) ed il sistema non riesce a risolvere la situazione.	EEXIST	<code>newpath</code> è una directory già esistente e non è vuota (anche <code>ENOTEMPTY</code>).	EINVAL	<code>newpath</code> contiene un prefisso di <code>oldpath</code> o più in generale si è cercato di creare una directory come sotto-directory di sé stessa.	EISDIR	<code>newpath</code> è una directory mentre <code>oldpath</code> non è una directory.	ENOTDIR	uno dei componenti dei <code>pathname</code> non è una directory o <code>oldpath</code> è una directory e <code>newpath</code> esiste e non è una directory.	EPERM	la directory contenente <code>oldpath</code> o quella contenente un <code>newpath</code> esistente hanno lo <i>sticky bit</i> e non si è i proprietari dei rispettivi file (o non si hanno privilegi amministrativi) oppure il filesystem non supporta l'operazione.	EXDEV	<code>oldpath</code> e <code>newpath</code> non sono sullo stesso filesystem e sotto lo stesso <i>mount point</i> .
EACCESS	manca il permesso di scrittura sulle directory contenenti <code>oldpath</code> e <code>newpath</code> o di attraversare il loro <code>pathname</code> o di scrivere su <code>newpath</code> se questa è una directory.																
EBUSY	<code>oldpath</code> o <code>newpath</code> sono in uso da parte di qualche processo (come directory di lavoro o come radice) o del sistema (come <i>mount point</i>) ed il sistema non riesce a risolvere la situazione.																
EEXIST	<code>newpath</code> è una directory già esistente e non è vuota (anche <code>ENOTEMPTY</code>).																
EINVAL	<code>newpath</code> contiene un prefisso di <code>oldpath</code> o più in generale si è cercato di creare una directory come sotto-directory di sé stessa.																
EISDIR	<code>newpath</code> è una directory mentre <code>oldpath</code> non è una directory.																
ENOTDIR	uno dei componenti dei <code>pathname</code> non è una directory o <code>oldpath</code> è una directory e <code>newpath</code> esiste e non è una directory.																
EPERM	la directory contenente <code>oldpath</code> o quella contenente un <code>newpath</code> esistente hanno lo <i>sticky bit</i> e non si è i proprietari dei rispettivi file (o non si hanno privilegi amministrativi) oppure il filesystem non supporta l'operazione.																
EXDEV	<code>oldpath</code> e <code>newpath</code> non sono sullo stesso filesystem e sotto lo stesso <i>mount point</i> .																

³³questo vale usando la *glibc*; nella *libc4* e nella *libc5* la funzione `remove` era un semplice alias alla funzione `unlink` e quindi non poteva essere usata per le directory.

³⁴la funzione è definita dallo standard ANSI C, ma si applica solo per i file, lo standard POSIX estende la funzione anche alle directory.

La funzione rinomina in `newpath` il file o la directory indicati dall'argomento `oldpath`. Il nome viene eliminato nella directory originale e ricreato nella directory di destinazione mantenendo il riferimento allo stesso *inode*. Non viene spostato nessun dato e l'*inode* del file non subisce nessuna modifica in quanto le modifiche sono eseguite sulle directory che contengono `newpath` ed `oldpath`.

Il vantaggio nell'uso di questa funzione al posto della chiamata successiva di `link` e `unlink` è che l'operazione è eseguita atomicamente, non c'è modifica, per quanto temporanea, al *link count* del file e non può esistere un istante in cui un altro processo possa trovare attivi entrambi i nomi per lo stesso file se la destinazione non esiste o in cui questa sparisca temporaneamente se già esiste.

Dato che opera in maniera analoga la funzione è soggetta alle stesse restrizioni di `link`, quindi è necessario che `oldpath` e `newpath` siano nello stesso filesystem e facciano riferimento allo stesso *mount point*, e che il filesystem supporti questo tipo di operazione. Qualora questo non avvenga si dovrà effettuare l'operazione in maniera non atomica copiando il file a destinazione e poi cancellando l'originale.

Il comportamento della funzione è diverso a seconda che si voglia rinominare un file o una directory. Se ci riferisce ad un file allora `newpath`, se esiste, non deve essere una directory, altrimenti si avrà un errore di `EISDIR`. Se `newpath` indica un file già esistente questo verrà rimpiazzato atomicamente, ma nel caso in cui `rename` fallisca il kernel assicura che esso non sarà toccato. I caso di sovrascrittura però potrà esistere una breve finestra di tempo in cui sia `oldpath` che `newpath` potranno fare entrambi riferimento al file che viene rinominato.

Se `oldpath` è una directory allora `newpath`, se esistente, deve essere una directory vuota, altrimenti si avranno gli errori `ENOTDIR` (se non è una directory) o `ENOTEMPTY` o `EEXIST` (se non è vuota). Chiaramente `newpath` non potrà contenere `oldpath` nel suo *pathname*, non essendo possibile rendere una directory una sottodirectory di sé stessa, se questo fosse il caso si otterrebbe un errore di `EINVAL`.

Se `oldpath` si riferisce ad un collegamento simbolico questo sarà rinominato restando tale senza nessun effetto sul file a cui fa riferimento. Se invece `newpath` esiste ed è un collegamento simbolico verrà cancellato come qualunque altro file. Infine qualora `oldpath` e `newpath` siano due nomi che già fanno riferimento allo stesso file lo standard POSIX prevede che la funzione non ritorni un errore, e semplicemente non faccia nulla, lasciando entrambi i nomi. Linux segue questo standard, anche se, come fatto notare dal manuale della *glibc*, il comportamento più ragionevole sarebbe quello di cancellare `oldpath`.

In tutti i casi si dovranno avere i permessi di scrittura nelle directory contenenti `oldpath` e `newpath`, e nel caso `newpath` sia una directory vuota già esistente anche su di essa (perché dovrà essere aggiornata la voce `“.”`). Se poi le directory contenenti `oldpath` o `newpath` hanno lo *sticky bit* attivo (vedi sez. 4.4.2) si dovrà essere i proprietari dei file (o delle directory) che si vogliono rinominare, o avere i permessi di amministratore.

4.2.2 La creazione e la cancellazione delle directory

Benché in sostanza le directory non siano altro che dei file contenenti elenchi di nomi con riferimenti ai rispettivi *inode*, non è possibile trattarle come file ordinari e devono essere create direttamente dal kernel attraverso una opportuna *system call*.³⁵ La funzione di sistema usata per creare una directory è `mkdir`, ed il suo prototipo è:

³⁵questo è quello che permette anche, attraverso l'uso del VFS, l'utilizzo di diversi formati per la gestione dei suddetti elenchi, dalle semplici liste a strutture complesse come alberi binari, hash, ecc. che consentono una ricerca veloce quando il numero di file è molto grande.

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *dirname, mode_t mode)
```

Crea una nuova directory.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES non c'è il permesso di scrittura per la directory in cui si vuole inserire la nuova directory o di attraversamento per le directory al di sopra di essa.

EEXIST un file o una directory o un collegamento simbolico con quel nome esiste già.

EMLINK la directory in cui si vuole creare la nuova directory contiene troppi file; sotto Linux questo normalmente non avviene perché il filesystem standard consente la creazione di un numero di file maggiore di quelli che possono essere contenuti nel disco, ma potendo avere a che fare anche con filesystem di altri sistemi questo errore può presentarsi.

ENOSPC non c'è abbastanza spazio sul file system per creare la nuova directory o si è esaurita la quota disco dell'utente.

ed inoltre **EFAULT**, **ELOOP**, **ENAMETOOLONG**, **ENOENT**, **ENOMEM**, **ENOTDIR**, **EPERM**, **EROFS** nel loro significato generico.

La funzione crea una nuova directory vuota, che contiene cioè solo le due voci standard presenti in ogni directory (“.” e “..”), con il nome indicato dall'argomento `dirname`.

I permessi di accesso (vedi sez. 4.4) con cui la directory viene creata sono specificati dall'argomento `mode`, i cui possibili valori sono riportati in tab. 4.12; si tenga presente che questi sono modificati dalla maschera di creazione dei file (si veda sez. 4.4.3). La titolarità della nuova directory è impostata secondo quanto illustrato in sez. 4.4.4.

Come accennato in precedenza per eseguire la cancellazione di una directory è necessaria una specifica funzione di sistema, `rmdir`, il suo prototipo è:

```
#include <sys/stat.h>
int rmdir(const char *dirname)
```

Cancella una directory.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES non c'è il permesso di scrittura per la directory che contiene la directory che si vuole cancellare, o non c'è il permesso di attraversare (esecuzione) una delle directory specificate in `dirname`.

EBUSY la directory specificata è la directory di lavoro o la radice di qualche processo o un *mount point*.

EINVAL si è usato “.” come ultimo componente di `dirname`.

EPERM il filesystem non supporta la cancellazione di directory, oppure la directory che contiene `dirname` ha lo *sticky bit* impostato e non si è i proprietari della directory o non si hanno privilegi amministrativi.

ed inoltre **EFAULT**, **ELOOP**, **ENAMETOOLONG**, **ENOENT**, **ENOMEM**, **ENOTDIR**, **ENOTEMPTY** e **EROFS** nel loro significato generico.

La funzione cancella la directory `dirname`, che deve essere vuota, la directory deve cioè contenere le due voci standard “.” e “..” e niente altro. Il nome può essere indicato con un *pathname* assoluto o relativo, ma si deve fare riferimento al nome nella directory genitrice, questo significa che *pathname* terminanti in “.” e “..” anche se validi in altri contesti, causeranno il fallimento della funzione.

Inoltre per eseguire la cancellazione, oltre ad essere vuota, occorre anche che la directory non sia utilizzata, questo vuol dire anche che non deve essere la directory di lavoro (vedi

sez. 4.2.4) o la directory radice (vedi sez. 4.5.4) di nessun processo, od essere utilizzata come *mount point*.

Se la directory cancellata risultasse aperta in qualche processo per una lettura dei suoi contenuti (vedi sez. 4.2.3), pur scomparendo dal filesystem e non essendo più possibile accedervi o crearvi altri file, le risorse ad essa associate verrebbero disallocate solo alla chiusura di tutti questi ulteriori riferimenti.

4.2.3 Lettura e scansione delle directory

Benché le directory alla fine non siano altro che dei file che contengono delle liste di nomi associati ai relativi *inode*, per il ruolo che rivestono nella struttura del sistema non possono essere trattate come dei normali file di dati. Ad esempio, onde evitare inconsistenze all'interno del filesystem, solo il kernel può scrivere il contenuto di una directory, e non può essere un processo a inserirvi direttamente delle voci con le usuali funzioni di scrittura.

Ma se la scrittura e l'aggiornamento dei dati delle directory è compito del kernel, sono molte le situazioni in cui i processi necessitano di poterne leggere il contenuto. Benché questo possa essere fatto direttamente (vedremo in sez. 5.1.2 che è possibile aprire una directory come se fosse un file, anche se solo in sola lettura) in generale il formato con cui esse sono scritte può dipendere dal tipo di filesystem, tanto che, come riportato in tab. 4.2, il VFS prevede una apposita funzione per la lettura delle directory.

Tutto questo si riflette nello standard POSIX³⁶ che ha introdotto una apposita interfaccia per la lettura delle directory, basata sui cosiddetti *directory stream*, chiamati così per l'analogia con i *file stream* dell'interfaccia standard ANSI C che vedremo in sez. 5.3. La prima funzione di questa interfaccia è *opendir*, il cui prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname)
```

Apre un *directory stream*.

La funzione ritorna un puntatore al *directory stream* in caso di successo e NULL per un errore, nel qual caso *errno* assumerà uno dei valori EACCES, EMFILE, ENFILE, ENOENT, ENOMEM e ENOTDIR nel loro significato generico.

La funzione apre un *directory stream* per la directory *dirname*, ritornando il puntatore ad un oggetto di tipo DIR (che è il tipo opaco usato dalle librerie per gestire i *directory stream*) da usare per tutte le operazioni successive, la funzione inoltre posiziona lo *stream* sulla prima voce contenuta nella directory.

Si tenga presente che comunque la funzione opera associando il *directory stream* ad un opportuno file descriptor (vedi sez. 5.1.1) sottostante, sul quale vengono compiute le operazioni. Questo viene sempre aperto impostando il flag di *close-on-exec* (si ricordi quanto detto in sez. 3.1.6), così da evitare che resti aperto in caso di esecuzione di un altro programma.

Nel caso in cui sia necessario conoscere il file descriptor associato ad un *directory stream* si può usare la funzione *dirfd*,³⁷ il cui prototipo è:

³⁶le funzioni erano presenti in SVr4 e 4.3BSD, la loro specifica è riportata in POSIX.1-2001.

³⁷questa funzione è una estensione introdotta con BSD 4.3-Reno ed è presente in Linux con le libc5 (a partire dalla versione 5.1.2) e con la *glibc* ma non presente in POSIX fino alla revisione POSIX.1-2008, per questo per poterla utilizzare fino alla versione 2.10 della *glibc* era necessario definire le macro `_BSD_SOURCE` o `_SVID_SOURCE`, dalla versione 2.10 si possono usare anche `_POSIX_C_SOURCE >= 200809L` o `_XOPEN_SOURCE >= 700`.


```
#include <sys/types.h>
#include <dirent.h>
int dirfd(DIR *dir)
```

Legge il file descriptor associato ad un *directory stream*.

La funzione ritorna un valore positivo corrispondente al file descriptor in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `dir` non è un puntatore ad un *directory stream*.

`ENOTSUP` l'implementazione non supporta l'uso di un file descriptor per la directory.

La funzione restituisce il file descriptor associato al *directory stream* `dir`. Di solito si utilizza questa funzione in abbinamento a funzioni che operano sui file descriptor, ad esempio si potrà usare `fstat` per ottenere le proprietà della directory, o `fchdir` per spostare su di essa la directory di lavoro (vedi sez. 4.2.4).

Viceversa se si è aperto un file descriptor corrispondente ad una directory è possibile associarvi un *directory stream* con la funzione `fdopendir`,³⁸ il cui prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
DIR *fdopendir(int fd)
```

Associa un *directory stream* ad un file descriptor.

La funzione ritorna un puntatore al *directory stream* in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori `EBADF` o `ENOMEM` nel loro significato generico.

La funzione è identica a `opendir`, ma ritorna un *directory stream* facendo riferimento ad un file descriptor `fd` che deve essere stato aperto in precedenza; la funzione darà un errore qualora questo non corrisponda ad una directory. L'uso di questa funzione permette di rispondere agli stessi requisiti delle funzioni “*at*” che vedremo in sez. 5.2.4.

Una volta utilizzata il file descriptor verrà usato internamente dalle funzioni che operano sul *directory stream* e non dovrà essere più utilizzato all'interno del proprio programma. In particolare dovrà essere chiuso attraverso il *directory stream* con `closedir` e non direttamente. Si tenga presente inoltre che `fdopendir` non modifica lo stato di un eventuale flag di *close-on-exec*, che pertanto dovrà essere impostato esplicitamente in fase di apertura del file descriptor.

Una volta che si sia aperto un *directory stream* la lettura del contenuto della directory viene effettuata attraverso la funzione `readdir`, il cui prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir)
```

Legge una voce dal *directory stream*.

La funzione ritorna il puntatore alla struttura contenente i dati in caso di successo e `NULL` per un errore o se si è raggiunta la fine dello *stream*. Il solo codice di errore restituito in `errno` è `EBADF` qualora `dir` non indichi un *directory stream* valido.

La funzione legge la voce corrente nella directory, posizionandosi sulla voce successiva. Pertanto se si vuole leggere l'intero contenuto di una directory occorrerà ripetere l'esecuzione della funzione fintanto che non si siano esaurite tutte le voci in essa presenti, che viene segnalata dalla restituzione di `NULL` come valore di ritorno. Si può distinguere questa condizione da un errore in quanto in questo caso `errno` non verrebbe modificata.

³⁸questa funzione è però disponibile solo a partire dalla versione 2.4 della *glibc*, ed è stata introdotta nello standard POSIX solo a partire dalla revisione POSIX.1-2008, prima della versione 2.10 della *glibc* per poterla utilizzare era necessario definire la macro `_GNU_SOURCE`, dalla versione 2.10 si possono usare anche `_POSIX_C_SOURCE >= 200809L` o `_XOPEN_SOURCE >= 700`.

I dati letti da `readdir` vengono memorizzati in una struttura `dirent`, la cui definizione è riportata in fig. 4.10.³⁹ La funzione non è rientrante e restituisce il puntatore ad una struttura allocata staticamente, che viene sovrascritta tutte le volte che si ripete la lettura di una voce sullo stesso *directory stream*.

Di questa funzione esiste anche una versione rientrante, `readdir_r`,⁴⁰ che non usa una struttura allocata staticamente, e può essere utilizzata anche con i *thread*, il suo prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
int readdir_r(DIR *dir, struct dirent *entry, struct dirent **result)
    Legge una voce dal directory stream.
```

La funzione ritorna 0 in caso di successo ed un numero positivo per un errore, nel qual caso `errno` assumerà gli stessi valori di `readdir`.

La funzione restituisce in `result` come *value result argument* l'indirizzo della struttura `dirent` dove sono stati salvati i dati, che deve essere allocata dal chiamante, ed il cui indirizzo deve essere indicato con l'argomento `entry`. Se si è raggiunta la fine del *directory stream* invece in `result` viene restituito il valore `NULL`.

```
struct dirent {
    ino_t d_ino;           /* inode number */
    off_t d_off;           /* offset to the next dirent */
    unsigned short int d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file;
                           by all file system types */
    char d_name[256];      /* filename */
};
```

Figura 4.10: La struttura `dirent` per la lettura delle informazioni dei file.

I vari campi di `dirent` contengono le informazioni relative alle voci presenti nella directory. Sia BSD che SVr4 che POSIX.1-2001⁴¹ prevedono che siano sempre presenti il campo `d_name`, che contiene il nome del file nella forma di una stringa terminata da uno zero, ed il campo `d_ino`, che contiene il numero di *inode* cui il file è associato e corrisponde al campo `st_ino` di `stat`. La presenza di ulteriori campi opzionali oltre i due citati è segnalata dalla definizione di altrettante macro nella forma `_DIRENT_HAVE_D_XXX` dove `XXX` è il nome del relativo campo. Come si può evincere da fig. 4.10 nel caso di Linux sono pertanto definite le macro `_DIRENT_HAVE_D_TYPE`, `_DIRENT_HAVE_D_OFF` e `_DIRENT_HAVE_D_RECLEN`, mentre non è definita la macro `_DIRENT_HAVE_D_NAMLEN`.

Dato che possono essere presenti campi opzionali e che lo standard POSIX.1-2001 non specifica una dimensione definita per il nome dei file (che può variare a seconda del filesystem), ma solo un limite superiore pari a `NAME_MAX` (vedi tab. 6.7), in generale per allocare una struttura `dirent` in maniera portabile occorre eseguire un calcolo per ottenere le dimensioni

³⁹la definizione è quella usata da Linux, che si trova nel file `/usr/include/bits/dirent.h`, essa non contempla la presenza del campo `d_namlen` che indica la lunghezza del nome del file.

⁴⁰per usarla è necessario definire una qualunque delle macro `_POSIX_C_SOURCE >= 1`, `_XOPEN_SOURCE`, `_BSD_SOURCE`, `_SVID_SOURCE`, `_POSIX_SOURCE`.

⁴¹il vecchio standard POSIX prevedeva invece solo la presenza del campo `d_fileno`, identico `d_ino`, che in Linux era definito come alias di quest'ultimo, mentre il campo `d_name` era considerato dipendente dall'implementazione.

appropriate per il proprio sistema.⁴² Lo standard però richiede che il campo `d_name` sia sempre l'ultimo della struttura, questo ci consente di ottenere la dimensione della prima parte con la macro di utilità generica `offsetof`, che si può usare con il seguente prototipo:

```
#include <stddef.h>
size_t offsetof(type, member)
```

Restituisce la posizione del campo `member` nella struttura `type`.

Ottenuta allora con `offsetof(struct dirent, d_name)` la dimensione della parte iniziale della struttura, basterà sommarci la dimensione massima dei nomi dei file nel filesystem che si sta usando, che si può ottenere attraverso la funzione `pathconf` (per la quale si rimanda a sez. 6.1.2) più un ulteriore carattere per la terminazione della stringa.

Per quanto riguarda il significato dei campi opzionali, il campo `d_type` indica il tipo di file (se *fifo*, directory, collegamento simbolico, ecc.), e consente di evitare una successiva chiamata a `lstat` (vedi sez. 4.3.1) per determinarlo. I suoi possibili valori sono riportati in tab. 4.5. Si tenga presente che questo valore è disponibile solo per i filesystem che ne supportano la restituzione (fra questi i più noti sono *Btrfs*, *ext2*, *ext3*, e *ext4*), per gli altri si otterrà sempre il valore `DT_UNKNOWN`.⁴³

Valore	Tipo di file
DT_UNKNOWN	Tipo sconosciuto.
DT_REG	File normale.
DT_DIR	Directory.
DT_LNK	Collegamento simbolico.
DT_FIFO	<i>Fifo</i> .
DT SOCK	Socket.
DT_CHR	Dispositivo a caratteri.
DT_BLK	Dispositivo a blocchi.

Tabella 4.5: Costanti che indicano i vari tipi di file nel campo `d_type` della struttura `dirent`.

Per la conversione da e verso l'analogo valore mantenuto dentro il campo `st_mode` di `stat` (vedi fig. 4.13) sono definite anche due macro di conversione, `IFTODT` e `DTTOIF`:

```
#include <dirent.h>
int IFTODT(mode_t MODE)
```

Converte il tipo di file dal formato di `st_mode` a quello di `d_type`.

```
mode_t DTTOIF(int DTYPE)
```

Converte il tipo di file dal formato di `d_type` a quello di `st_mode`.

Il campo `d_off` contiene invece la posizione della voce successiva della directory, mentre il campo `d_reclen` la lunghezza totale della voce letta. Con questi due campi diventa possibile, determinando la posizione delle varie voci, spostarsi all'interno dello *stream* usando la funzione `seekdir`,⁴⁴ il cui prototipo è:

```
#include <dirent.h>
void seekdir(DIR *dir, off_t offset)
```

Cambia la posizione all'interno di un *directory stream*.

La funzione non ritorna niente e non imposta errori.

⁴²in SVr4 la lunghezza del campo è definita come `NAME_MAX+1` che di norma porta al valore di 256 byte usato anche in fig. 4.10.

⁴³inoltre fino alla versione 2.1 della *glibc*, pur essendo il campo `d_type` presente, il suo uso non era implementato, e veniva restituito comunque il valore `DT_UNKNOWN`.

⁴⁴sia questa funzione che `telldir`, sono estensioni prese da BSD, ed introdotte nello standard POSIX solo a partire dalla revisione POSIX.1-2001, per poterle utilizzare deve essere definita una delle macro `_XOPEN_SOURCE`, `_BSD_SOURCE` o `_SVID_SOURCE`.

La funzione non ritorna nulla e non segnala errori, è però necessario che il valore dell'argomento *offset* sia valido per lo *stream* *dir*; esso pertanto deve essere stato ottenuto o dal valore di *d_off* di *dirent* o dal valore restituito dalla funzione *tellldir*, che legge la posizione corrente; il cui prototipo è:⁴⁵

```
#include <dirent.h>
long tellldir(DIR *dir)
```

Ritorna la posizione corrente in un *directory stream*.

La funzione ritorna la posizione corrente nello *stream* (un numero positivo) in caso di successo e -1 per un errore, nel qual caso *errno* assume solo il valore di *EBADF*, corrispondente ad un valore errato per *dir*.

La sola funzione di posizionamento per un *directory stream* prevista originariamente dallo standard POSIX è *rewinddir*, che riporta la posizione a quella iniziale; il suo prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dir)
```

Si posiziona all'inizio di un *directory stream*.

La funzione non ritorna niente e non imposta errori.

Una volta completate le operazioni si può chiudere il *directory stream*, ed il file descriptor ad esso associato, con la funzione *closedir*, il cui prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dir)
```

Chiude un *directory stream*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assume solo il valore *EBADF*.

A parte queste funzioni di base in BSD 4.3 venne introdotta un'altra funzione che permette di eseguire una scansione completa, con tanto di ricerca ed ordinamento, del contenuto di una directory; la funzione è *scandir*⁴⁶ ed il suo prototipo è:

```
#include <dirent.h>
int scandir(const char *dir, struct dirent ***namelist,
            int(*filter)(const struct dirent *),
            int(*compar)(const struct dirent **, const struct dirent **))
```

Esegue una scansione di un *directory stream*.

La funzione ritorna il numero di voci trovate in caso di successo e -1 per un errore, nel qual caso *errno* può assumere solo il valore *ENOMEM*.

Al solito, per la presenza fra gli argomenti di due puntatori a funzione, il prototipo non è molto comprensibile; queste funzioni però sono quelle che controllano rispettivamente la selezione di una voce, passata con l'argomento *filter*, e l'ordinamento di tutte le voci selezionate, specificata dell'argomento *compar*.

La funzione legge tutte le voci della directory indicata dall'argomento *dir*, passando un puntatore a ciascuna di esse (una struttura *dirent*) come argomento della funzione di selezione specificata da *filter*; se questa ritorna un valore diverso da zero il puntatore viene inserito in

⁴⁵prima della *glibc* 2.1.1 la funzione restituiva un valore di tipo *off_t*, sostituito a partire dalla versione 2.1.2 da *long* per conformità a POSIX.1-2001.

⁴⁶in Linux questa funzione è stata introdotta fin dalle *libc4* e richiede siano definite le macro *_BSD_SOURCE* o *_SVID_SOURCE*.

un vettore che viene allocato dinamicamente con `malloc`. Qualora si specifichi un valore `NULL` per l'argomento `filter` non viene fatta nessuna selezione e si ottengono tutte le voci presenti.

Le voci selezionate possono essere riordinate tramite `qsort`, le modalità del riordinamento possono essere personalizzate usando la funzione `compar` come criterio di ordinamento di `qsort`, la funzione prende come argomenti le due strutture `dirent` da confrontare restituendo un valore positivo, nullo o negativo per indicarne l'ordinamento; alla fine l'indirizzo della lista ordinata dei puntatori alle strutture `dirent` viene restituito nell'argomento `namelist`.⁴⁷

Per l'ordinamento, vale a dire come valori possibili per l'argomento `compar`, sono disponibili due funzioni predefinite, `alphasort` e `versionsort`, i cui prototipi sono:

```
#include <dirent.h>
int alphasort(const void *a, const void *b)
int versionsort(const void *a, const void *b)
```

Riordinano le voci di *directory stream*.

Le funzioni restituiscono un valore minore, uguale o maggiore di zero qualora il primo argomento sia rispettivamente minore, uguale o maggiore del secondo e non forniscono errori.

La funzione `alphasort` deriva da BSD ed è presente in Linux fin dalle *libc4*⁴⁸ e deve essere specificata come argomento `compar` per ottenere un ordinamento alfabetico secondo il valore del campo `d_name` delle varie voci. La *glibc* prevede come estensione⁴⁹ anche `versionsort`, che ordina i nomi tenendo conto del numero di versione, cioè qualcosa per cui `file10` viene comunque dopo `file4`.

Un semplice esempio dell'uso di queste funzioni è riportato in fig. 4.11, dove si è riportata la sezione principale di un programma che, usando la funzione di scansione illustrata in fig. 4.12, stampa i nomi dei file contenuti in una directory e la relativa dimensione, in sostanza una versione semplificata del comando `ls`.

Il programma è estremamente semplice; in fig. 4.11 si è omessa la parte di gestione delle opzioni, che prevede solo l'uso di una funzione per la stampa della sintassi, anch'essa omessa, ma il codice completo può essere trovato coi sorgenti allegati alla guida nel file `myls.c`.

In sostanza tutto quello che fa il programma, dopo aver controllato (12-15) di avere almeno un argomento, che indicherà la directory da esaminare, è chiamare (16) la funzione `dir_scan` per eseguire la scansione, usando la funzione `do_ls` (22-29) per fare tutto il lavoro.

Quest'ultima si limita (26) a chiamare `stat` sul file indicato dalla directory entry passata come argomento (il cui nome è appunto `dirent->d_name`), memorizzando in una opportuna struttura `data` i dati ad esso relativi, per poi provvedere (27) a stampare il nome del file e la dimensione riportata in `data`.

Dato che la funzione verrà chiamata all'interno di `dir_scan` per ogni voce presente, questo è sufficiente a stampare la lista completa dei file e delle relative dimensioni. Si noti infine come si restituisca sempre 0 come valore di ritorno per indicare una esecuzione senza errori.

Tutto il grosso del lavoro è svolto dalla funzione `dir_scan`, riportata in fig. 4.12. La funzione è volutamente generica e permette di eseguire una funzione, passata come secondo argomento, su tutte le voci di una directory. La funzione inizia con l'aprire (18-22) uno *stream* sulla directory passata come primo argomento, stampando un messaggio in caso di errore.

⁴⁷la funzione alloca automaticamente la lista, e restituisce, come *value result argument*, l'indirizzo della stessa; questo significa che `namelist` deve essere dichiarato come `struct dirent **namelist` ed alla funzione si deve passare il suo indirizzo.

⁴⁸la versione delle *libc4* e *libc5* usa però come argomenti dei puntatori a delle strutture `dirent`; la *glibc* usa il prototipo originario di BSD, mostrato anche nella definizione, che prevede puntatori a `void`.

⁴⁹la *glibc*, a partire dalla versione 2.1, effettua anche l'ordinamento alfabetico tenendo conto delle varie localizzazioni, usando `strcoll` al posto di `strcmp`.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <dirent.h>          /* directory */
4 #include <stdlib.h>          /* C standard library */
5 #include <unistd.h>
6 /* computation function for dir_scan */
7 int do_ls(struct dirent * direntry);
8 /* main body */
9 int main(int argc, char *argv[])
10 {
11     ...
12     if ((argc - optind) != 1) {          /* There must be remaing parameters */
13         printf("Wrong_number_of_arguments_%d\n", argc - optind);
14         usage();
15     }
16     dir_scan(argv[1], do_ls);
17     exit(0);
18 }
19 /*
20  * Routine to print file name and size inside dir_scan
21  */
22 int do_ls(struct dirent * direntry)
23 {
24     struct stat data;
25
26     stat(direntry->d_name, &data);      /* get stat data */
27     printf("File:_%s_\t_size:_%d\n", direntry->d_name, data.st_size);
28     return 0;
29 }

```

Figura 4.11: Esempio di codice per eseguire la lista dei file contenuti in una directory.

Il passo successivo (23-24) è cambiare directory di lavoro (vedi sez. 4.2.4), usando in sequenza le funzioni `dirfd` e `fchdir` (in realtà si sarebbe potuto usare direttamente `chdir` su `dirname`), in modo che durante il successivo ciclo (26-30) sulle singole voci dello *stream* ci si trovi all'interno della directory.⁵⁰

Avendo usato lo stratagemma di fare eseguire tutte le manipolazioni necessarie alla funzione passata come secondo argomento, il ciclo di scansione della directory è molto semplice; si legge una voce alla volta (26) all'interno di una istruzione di `while` e fintanto che si riceve una voce valida, cioè un puntatore diverso da `NULL`, si esegue (27) la funzione di elaborazione `compars` (che nel nostro caso sarà `do_ls`), ritornando con un codice di errore (28) qualora questa presenti una anomalia, identificata da un codice di ritorno negativo. Una volta terminato il ciclo la funzione si conclude con la chiusura (31) dello *stream*⁵¹ e la restituzione (32) del codice di operazioni concluse con successo.

⁵⁰questo è essenziale al funzionamento della funzione `do_ls`, e ad ogni funzione che debba usare il campo `d_name`, in quanto i nomi dei file memorizzati all'interno di una struttura `dirent` sono sempre relativi alla directory in questione, e senza questo posizionamento non si sarebbe potuto usare `stat` per ottenere le dimensioni.

⁵¹nel nostro caso, uscendo subito dopo la chiamata, questo non servirebbe, in generale però l'operazione è necessaria, dato che la funzione può essere invocata molte volte all'interno dello stesso processo, per cui non chiudere i *directory stream* comporterebbe un consumo progressivo di risorse, con conseguente rischio di esaurimento delle stesse.

```

1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <dirent.h>          /* directory */
4 #include <stdlib.h>          /* C standard library */
5 #include <unistd.h>
6
7 /*
8  * Function dir_scan:
9  *
10  * Input:  the directory name and a computation function
11  * Return: 0 if OK, -1 on errors
12  */
13 int dir_scan(char * dirname, int(*compute)(struct dirent *))
14 {
15     DIR * dir;
16     struct dirent *dirent;
17
18     if ( (dir = opendir(dirname)) == NULL) { /* open directory */
19         printf("Opening_%s\n", dirname);    /* on error print messages */
20         perror("Cannot_open_directory");    /* and then return */
21         return -1;
22     }
23     fd = dirfd(dir);                        /* get file descriptor */
24     fchdir(fd);                             /* change directory */
25     /* loop on directory entries */
26     while ( (dirent = readdir(dir)) != NULL) { /* read entry */
27         if (compute(dirent)) {              /* execute function on it */
28             return -1;                      /* on error return */
29         }
30     }
31     closedir(dir);
32     return 0;
33 }

```

Figura 4.12: Codice della funzione di scansione di una directory contenuta nel file `dir_scan.c`.

4.2.4 La directory di lavoro

Come accennato in sez. 3.1.3 a ciascun processo è associata una directory nell'albero dei file,⁵² che è chiamata *directory corrente* o *directory di lavoro* (in inglese *current working directory*). La directory di lavoro è quella da cui si parte quando un *pathname* è espresso in forma relativa, dove il “relativa” fa riferimento appunto a questa directory.

Quando un utente effettua il login, questa directory viene impostata alla *home directory* del suo account. Il comando `cd` della shell consente di cambiarla a piacere, spostandosi da una directory ad un'altra, il comando `pwd` la stampa sul terminale. Siccome la directory di lavoro resta la stessa quando viene creato un processo figlio (vedi sez. 3.1.3), la directory di lavoro della shell diventa anche la directory di lavoro di qualunque comando da essa lanciato.

Dato che è il kernel che tiene traccia dell'*inode* della directory di lavoro di ciascun processo, per ottenerne il *pathname* occorre usare una apposita funzione, `getcwd`,⁵³ il cui prototipo è:

⁵²questa viene mantenuta all'interno dei dati della sua *task_struct* (vedi fig. 3.2), più precisamente nel campo `pwd` della sotto-struttura *fs_struct*.

⁵³con Linux `getcwd` è una *system call* dalla versione 2.1.9, in precedenza il valore doveva essere ottenuto tramite il filesystem `/proc` da `/proc/self/cwd`.

```
#include <unistd.h>
char *getcwd(char *buffer, size_t size)
```

Legge il *pathname* della directory di lavoro corrente.

La funzione ritorna il puntatore a *buffer* in caso di successo e NULL per un errore, nel qual caso *errno* assumerà uno dei valori:

EACCES manca il permesso di lettura o di attraversamento su uno dei componenti del *pathname* (cioè su una delle directory superiori alla corrente).

EINVAL l'argomento *size* è zero e *buffer* non è nullo.

ENOENT la directory di lavoro è stata eliminata.

ERANGE l'argomento *size* è più piccolo della lunghezza del *pathname*.

ed inoltre EFAULT ed ENOMEM nel loro significato generico.

La funzione restituisce il *pathname* completo della directory di lavoro corrente nella stringa puntata da *buffer*, che deve essere precedentemente allocata, per una dimensione massima di *size*. Il buffer deve essere sufficientemente largo da poter contenere il *pathname* completo più lo zero di terminazione della stringa. Qualora esso ecceda le dimensioni specificate con *size* la funzione restituisce un errore.

A partire dal kernel Linux 2.6.36 il nome può avere come prefisso la stringa (*unreachable*) se la directory di lavoro resta fuori dalla directory radice del processo dopo un *chroot* (torneremo su questi argomenti in sez. 4.5.4); pertanto è sempre opportuno controllare il primo carattere della stringa restituita dalla funzione per evitare di interpretare male un *pathname* irraggiungibile.

Come estensione allo standard POSIX.1, supportata da Linux e dalla *glibc*, si può anche specificare un puntatore nullo come *buffer* nel qual caso la stringa sarà allocata automaticamente per una dimensione pari a *size* qualora questa sia diversa da zero, o della lunghezza esatta del *pathname* altrimenti. In questo caso ci si deve ricordare di disallocare la stringa con *free* una volta cessato il suo utilizzo.

Un uso comune di *getcwd* è quello di salvarsi la directory di lavoro all'avvio del programma per poi potervi tornare in un tempo successivo, un metodo alternativo più veloce, se non si è a corto di file descriptor, è invece quello di aprire all'inizio la directory corrente (vale a dire “.”) e tornarvi in seguito con *chdir*.

Di questa funzione esiste una versione alternativa per compatibilità all'indietro con BSD, *getwd*, che non prevede l'argomento *size* e quindi non consente di specificare la dimensione di *buffer* che dovrebbe essere allocato in precedenza ed avere una dimensione sufficiente (per BSD maggiore *PATH_MAX*, che di solito 256 byte, vedi sez. 6.1.1). Il problema è che su Linux non esiste una dimensione superiore per la lunghezza di un *pathname*, per cui non è detto che il buffer sia sufficiente a contenere il nome del file, e questa è la ragione principale per cui questa funzione è deprecata, e non la tratteremo.

Una seconda funzione usata per ottenere la directory di lavoro è *get_current_dir_name* (la funzione è una estensione GNU e presente solo nella *glibc*) che non prende nessun argomento ed è sostanzialmente equivalente ad una *getcwd*(NULL, 0), con la differenza che se disponibile essa ritorna il valore della variabile di ambiente *PWD*, che essendo costruita dalla shell può contenere un *pathname* comprendente anche dei collegamenti simbolici. Usando *getcwd* infatti, essendo il *pathname* ricavato risalendo all'indietro l'albero della directory, si perderebbe traccia di ogni passaggio attraverso eventuali collegamenti simbolici.

Per cambiare la directory di lavoro si può usare la funzione di sistema *chdir*, equivalente del comando di shell *cd*, il cui nome sta appunto per *change directory*, il suo prototipo è:

```
#include <unistd.h>
int chdir(const char *pathname)
```

Cambia la directory di lavoro per *pathname*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES manca il permesso di ricerca su uno dei componenti di `pathname`.

ENAMETOOLONG il nome indicato in `path` è troppo lungo.

ENOTDIR non si è specificata una directory.

ed inoltre EFAULT, EIO, ELOOP, ENOENT e ENOMEM nel loro significato generico.

La funzione cambia la directory di lavoro in `pathname` ed ovviamente `pathname` deve indicare una directory per la quale si hanno i permessi di accesso.

Dato che ci si può riferire ad una directory anche tramite un file descriptor, per cambiare directory di lavoro è disponibile anche la funzione di sistema `fchdir`, il cui prototipo è:

```
#include <unistd.h>
int fchdir(int fd)
```

Cambia la directory di lavoro per file descriptor.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori EBADF o EACCES nel loro significato generico.

La funzione è identica a `chdir`, ma prende come argomento un file descriptor `fd` invece di un `pathname`. Anche in questo caso `fd` deve essere un file descriptor valido che fa riferimento ad una directory. Inoltre l'unico errore di accesso possibile (tutti gli altri sarebbero occorsi all'apertura di `fd`), è quello in cui il processo non ha il permesso di attraversamento alla directory specificata da `fd`.

4.2.5 La creazione dei file speciali

Finora abbiamo parlato esclusivamente di file, directory e collegamenti simbolici, ma in sez. 1.2.3 abbiamo visto che il sistema prevede anche degli altri tipi di file, che in genere vanno sotto il nome generico di *file speciali*, come i file di dispositivo, le *fifo* ed i socket.

La manipolazione delle caratteristiche di questi file speciali, il cambiamento di nome o la loro cancellazione può essere effettuata con le stesse funzioni che operano sugli altri file, ma quando li si devono creare sono necessarie, come per le directory, delle funzioni apposite. La prima di queste è la funzione di sistema `mknod`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod(const char *pathname, mode_t mode, dev_t dev)
```

Crea un file speciale sul filesystem.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EEXIST `pathname` esiste già o è un collegamento simbolico.

EINVAL il valore di `mode` non indica un file, una *fifo*, un socket o un dispositivo.

EPERM non si hanno privilegi sufficienti a creare l'inode, o il filesystem su cui si è cercato di creare `pathname` non supporta l'operazione.

ed inoltre EACCES, EFAULT, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR e EROFS nel loro significato generico.

La funzione permette di creare un *inode* di tipo generico sul filesystem, e viene in genere utilizzata per creare i file di dispositivo, ma si può usare anche per creare qualunque tipo di file speciale ed anche file regolari. L'argomento `mode` specifica sia il tipo di file che si vuole

creare che i relativi permessi, secondo i valori riportati in tab. 4.7, che vanno combinati con un OR aritmetico. I permessi sono comunque modificati nella maniera usuale dal valore di *umask* (si veda sez. 4.4.3).

Per il tipo di file può essere specificato solo uno fra i seguenti valori: `S_IFREG` per un file regolare (che sarà creato vuoto), `S_IFBLK` per un dispositivo a blocchi, `S_IFCHR` per un dispositivo a caratteri, `S_IFSOCK` per un socket e `S_IFIFO` per una *fifo*;⁵⁴ un valore diverso comporterà l'errore `EINVAL`. Inoltre *pathname* non deve esistere, neanche come collegamento simbolico.

Qualora si sia specificato in *mode* un file di dispositivo (vale a dire o `S_IFBLK` o `S_IFCHR`), il valore di *dev* dovrà essere usato per indicare a quale dispositivo si fa riferimento, altrimenti il suo valore verrà ignorato. Solo l'amministratore può creare un file di dispositivo usando questa funzione (il processo deve avere la capacità `CAP_MKNOD`), ma in Linux⁵⁵ l'uso per la creazione di un file ordinario, di una *fifo* o di un socket è consentito anche agli utenti normali.

Gli *inode* creati con `mknod` apparterranno al proprietario e al gruppo del processo (usando *UID* e *GID* del gruppo effettivo) che li ha creati a meno non sia presente il bit *sgid* per la directory o sia stata attivata la semantica BSD per il filesystem (si veda sez. 4.4.4) in cui si va a creare l'*inode*, nel qual caso per il gruppo verrà usato il *GID* del proprietario della directory.

Nella creazione di un file di dispositivo occorre poi specificare correttamente il valore di *dev*; questo infatti è di tipo `dev_t`, che è un tipo primitivo (vedi tab. 1.2) riservato per indicare un numero di dispositivo. Il kernel infatti identifica ciascun dispositivo con un valore numerico, originariamente questo era un intero a 16 bit diviso in due parti di 8 bit chiamate rispettivamente *major number* e *minor number*, che sono poi i due numeri mostrati dal comando `ls -l` al posto della dimensione quando lo si esegue su un file di dispositivo.

Il *major number* identifica una classe di dispositivi (ad esempio la seriale, o i dischi IDE) e serve in sostanza per indicare al kernel quale è il modulo che gestisce quella classe di dispositivi. Per identificare uno specifico dispositivo di quella classe (ad esempio una singola porta seriale, o uno dei dischi presenti) si usa invece il *minor number*. L'elenco aggiornato di questi numeri con le relative corrispondenze ai vari dispositivi può essere trovato nel file `Documentation/devices.txt` allegato alla documentazione dei sorgenti del kernel.

Data la crescita nel numero di dispositivi supportati, ben presto il limite massimo di 256 si è rivelato troppo basso, e nel passaggio dal kernel della serie 2.4 alla serie 2.6 è stata aumentata a 32 bit la dimensione del tipo `dev_t`, con delle dimensioni passate a 12 bit per il *major number* e 20 bit per il *minor number*. La transizione però ha comportato il fatto che `dev_t` è diventato un tipo opaco, e la necessità di specificare il numero tramite delle opportune macro, così da non avere problemi di compatibilità con eventuali ulteriori estensioni.

Le macro sono definite nel file `sys/sysmacros.h`,⁵⁶ che viene automaticamente incluso quando si include `sys/types.h`. Si possono pertanto ottenere i valori del *major number* e *minor number* di un dispositivo rispettivamente con le macro `major` e `minor`:

⁵⁴con Linux la funzione non può essere usata per creare directory o collegamenti simbolici, si dovranno usare le funzioni `mkdir` e `symlink` a questo dedicate.

⁵⁵questo è un comportamento specifico di Linux, la funzione non è prevista dallo standard POSIX.1 originale, mentre è presente in SVr4 e 4.4BSD, ma esistono differenze nei comportamenti e nei codici di errore, tanto che questa è stata introdotta in POSIX.1-2001 con una nota che la definisce portabile solo quando viene usata per creare delle *fifo*, ma comunque deprecata essendo utilizzabile a tale scopo la specifica `mkfifo`.

⁵⁶se si usa la *glibc* dalla versione 2.3.3 queste macro sono degli alias alle versioni specifiche di questa libreria, `gnu_dev_major`, `gnu_dev_minor` e `gnu_dev_makedev` che si possono usare direttamente, al costo di una minore portabilità.

#include <sys/types.h> int major(dev_t dev)	Restituisce il <i>major number</i> del dispositivo dev.
int minor(dev_t dev)	Restituisce il <i>minor number</i> del dispositivo dev.

mentre una volta che siano noti *major number* e *minor number* si potrà costruire il relativo identificativo con la macro `makedev`:

#include <sys/types.h> dev_t makedev(int major, int minor)	Dati <i>major number</i> e <i>minor number</i> restituisce l'identificativo di un dispositivo.
---	--

Dato che la funzione di sistema `mknod` presenta diverse varianti nei vari sistemi unix-like, lo standard POSIX.1-2001 la dichiara portabile solo in caso di creazione delle *fifo*, ma anche in questo caso alcune combinazioni degli argomenti restano non specificate, per cui nello stesso standard è stata introdotta una funzione specifica per creare una *fifo* deprecando l'uso di `mknod` a tale riguardo. La funzione è `mkfifo` ed il suo prototipo è:

#include <sys/types.h> #include <sys/stat.h> int mkfifo(const char *pathname, mode_t mode)	Crea una <i>fifo</i> .
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà <code>EACCES</code> , <code>EEXIST</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> , <code>ENOSPC</code> , <code>ENOTDIR</code> e <code>EROFS</code> nel loro significato generico.	

La funzione crea la *fifo* `pathname` con i permessi `mode`. Come per `mknod` il file `pathname` non deve esistere (neanche come collegamento simbolico); al solito i permessi specificati da `mode` vengono modificati dal valore di `umask` (vedi sez. 4.4.3).

4.2.6 I file temporanei

In molte occasioni è utile poter creare dei file temporanei; benché la cosa sembri semplice, in realtà il problema è più sottile di quanto non appaia a prima vista. Infatti anche se sembrerebbe banale generare un nome a caso e creare il file dopo aver controllato che questo non esista, nel momento fra il controllo e la creazione si ha giusto lo spazio per una possibile *race condition* (si ricordi quanto visto in sez. 3.4.2).

Molti problemi di sicurezza derivano proprio da una creazione non accorta di file temporanei che lascia aperta questa *race condition*. Un attaccante allora potrà sfruttarla con quello che viene chiamato “*symlink attack*” dove nell'intervallo fra la generazione di un nome e l'accesso allo stesso, viene creato un collegamento simbolico con quel nome verso un file diverso, ottenendo, se il programma sotto attacco ne ha la capacità, un accesso privilegiato.⁵⁷

La *glibc* provvede varie funzioni per generare nomi di file temporanei, di cui si abbia certezza di unicità al momento della generazione; storicamente la prima di queste funzioni create a questo scopo era `tmpnam`,⁵⁸ il cui prototipo è:

#include <stdio.h> char *tmpnam(char *string)	Genera un nome univoco per un file temporaneo.
La funzione ritorna il puntatore alla stringa con il nome in caso di successo e NULL in caso di fallimento, non sono definiti errori.	

⁵⁷dal kernel 3.6 sono state introdotte delle contromisure, illustrate in sez. 9.1.3, che rendono impraticabili questo tipo di attacchi, ma questa non è una buona scusa per ignorare il problema.

⁵⁸la funzione è stata deprecata nella revisione POSIX.1-2008 dello standard POSIX.

La funzione restituisce il puntatore ad una stringa contenente un nome di file valido e non esistente al momento dell'invocazione. Se si è passato come argomento `string` un puntatore non nullo ad un buffer di caratteri questo deve essere di dimensione `L_tmpnam` ed il nome generato vi verrà copiato automaticamente, altrimenti il nome sarà generato in un buffer statico interno che verrà sovrascritto ad una chiamata successiva. Successive invocazioni della funzione continueranno a restituire nomi unici fino ad un massimo di `TMP_MAX` volte, limite oltre il quale il comportamento è indefinito. Al nome viene automaticamente aggiunto come prefisso la directory specificata dalla costante `P_tmpdir`.⁵⁹

Di questa funzione esiste una versione rientrante, `tmpnam_r`, che non fa nulla quando si passa `NULL` come argomento. Una funzione simile, `tempnam`, permette di specificare un prefisso per il file esplicitamente, il suo prototipo è:

```
#include <stdio.h>
char *tempnam(const char *dir, const char *pfx)
                                     Genera un nome univoco per un file temporaneo.
```

La funzione ritorna il puntatore alla stringa con il nome in caso di successo e `NULL` per un errore, nel qual caso `errno` potrà assumere solo il valore `ENOMEM` qualora fallisca l'allocazione della stringa.

La funzione alloca con `malloc` la stringa in cui restituisce il nome, per cui è sempre rientrante, occorre però ricordarsi di disallocare con `free` il puntatore che restituisce. L'argomento `pfx` specifica un prefisso di massimo 5 caratteri per il nome provvisorio. La funzione assegna come directory per il file temporaneo, verificando che esista e sia accessibile, la prima valida fra le seguenti:

- la variabile di ambiente `TMPDIR` (non ha effetto se non è definita o se il programma chiamante è *suid* o *sgid*, vedi sez. 4.4.2),
- il valore dell'argomento `dir` (se diverso da `NULL`),
- il valore della costante `P_tmpdir`,
- la directory `/tmp`.

In ogni caso, anche se con queste funzioni la generazione del nome è casuale, ed è molto difficile ottenere un nome duplicato, nulla assicura che un altro processo non possa avere creato, fra l'ottenimento del nome e l'apertura del file, un altro file o un collegamento simbolico con lo stesso nome. Per questo motivo quando si usa il nome ottenuto da una di queste funzioni occorre sempre assicurarsi che non si stia usando un collegamento simbolico e aprire il nuovo file in modalità di esclusione (cioè con l'opzione `O_EXCL` per i file descriptor o con il flag "x" per gli *stream*) che fa fallire l'apertura in caso il file sia già esistente. Essendo disponibili alternative migliori l'uso di queste funzioni è deprecato.

Per evitare di dovere effettuare a mano tutti questi controlli, lo standard POSIX definisce la funzione `tmpfile`, che permette di ottenere in maniera sicura l'accesso ad un file temporaneo, il suo prototipo è:

```
#include <stdio.h>
FILE *tmpfile(void)
                                     Apre un file temporaneo in lettura/scrittura.
```

La funzione ritorna il puntatore allo *stream* associato al file temporaneo in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EEXIST` non è stato possibile generare un nome univoco.

`EINTR` la funzione è stata interrotta da un segnale.

ed inoltre `EFAULT`, `EMFILE`, `ENFILE`, `ENOSPC`, `EROFS` e `EACCES` nel loro significato generico.

⁵⁹Le costanti `L_tmpnam`, `P_tmpdir` e `TMP_MAX` sono definite in `stdio.h`.

La funzione restituisce direttamente uno *stream* già aperto (in modalità *w+b*, si veda sez. 5.3.3) e pronto per l'uso, che viene automaticamente cancellato alla sua chiusura o all'uscita dal programma. Lo standard non specifica in quale directory verrà aperto il file, ma la *glibc* prima tenta con *P_tmpdir* e poi con */tmp*. Questa funzione è rientrante e non soffre di problemi di *race condition*.

Alcune versioni meno recenti di Unix non supportano queste funzioni; in questo caso si possono usare le vecchie funzioni *mktemp* e *mkstemp* che modificano una stringa di input che serve da modello e che deve essere conclusa da 6 caratteri "X" che verranno sostituiti da un codice unico. La prima delle due è analoga a *tmpnam* e genera soltanto un nome casuale, il suo prototipo è:

<pre>#include <stdlib.h> char *mktemp(char *template)</pre>	Genera un nome univoco per un file temporaneo.
<p>La funzione ritorna il puntatore a <i>template</i> in caso di successo e NULL per un errore, nel qual caso <i>errno</i> assumerà uno dei valori:</p> <p>EINVAL <i>template</i> non termina con XXXXXX.</p>	

La funzione genera un nome univoco sostituendo le XXXXXX finali di *template*; dato che *template* deve poter essere modificata dalla funzione non si può usare una stringa costante. Tutte le avvertenze riguardo alle possibili *race condition* date per *tmpnam* continuano a valere; inoltre in alcune vecchie implementazioni il valore usato per sostituire le XXXXXX viene formato con il *PID* del processo più una lettera, il che mette a disposizione solo 26 possibilità diverse per il nome del file, e rende il nome temporaneo facile da indovinare. Per tutti questi motivi la funzione è deprecata e non dovrebbe mai essere usata.

La seconda funzione, *mkstemp* è sostanzialmente equivalente a *tmpfile*, ma restituisce un file descriptor invece di un nome; il suo prototipo è:

<pre>#include <stdlib.h> int mkstemp(char *template)</pre>	Apri un file temporaneo.
<p>La funzione ritorna il file descriptor in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà uno dei valori:</p> <p>EEXIST non è riuscita a creare un file temporaneo, il contenuto di <i>template</i> è indefinito.</p> <p>EINVAL <i>template</i> non termina con XXXXXX.</p>	

Come per *mktemp* anche in questo caso *template* non può essere una stringa costante. La funzione apre un file in lettura/scrittura con la funzione *open*, usando l'opzione *O_EXCL* (si veda sez. 5.1.2), in questo modo al ritorno della funzione si ha la certezza di essere stati i creatori del file, i cui permessi (si veda sez. 4.4.1) sono impostati al valore *0600* (lettura e scrittura solo per il proprietario).⁶⁰ Di questa funzione esiste una variante *mkostemp*, introdotta specificamente dalla *glibc*,⁶¹ il cui prototipo è:

<pre>#include <stdlib.h> int mkostemp(char *template, int flags)</pre>	Apri un file temporaneo.
<p>La funzione ritorna un file descriptor in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà gli stessi valori di <i>mkstemp</i>.</p>	

⁶⁰questo è vero a partire dalla *glibc* 2.0.7, le versioni precedenti della *glibc* e le vecchie *libc5* e *libc4* usavano il valore *0666* che permetteva a chiunque di leggere e scrivere i contenuti del file.

⁶¹la funzione è stata introdotta nella versione 2.7 delle librerie e richiede che sia definita la macro *_GNU_SOURCE*.

la cui sola differenza è la presenza dell'ulteriore argomento `flags` che consente di specificare alcuni ulteriori flag (come `O_APPEND`, `O_CLOEXEC`, `O_SYNC`, il cui significato vedremo in sez. 5.1.2) da passare ad `open` nell'apertura del file.⁶²

Di queste due funzioni sono state poi introdotte, a partire dalla *glibc* 2.11 due varianti, `mkstemp` e `mkostemp`, che consentono di indicare anche un suffisso, i loro prototipi sono:

<pre>#include <stdlib.h> int mkstemp(char *template, int suffixlen)</pre>	<p>Apri un file temporaneo.</p>
<pre>int mkostemp(char *template, int suffixlen, int flags)</pre>	<p>Apri un file temporaneo.</p>
<p>Le funzioni hanno gli stessi valori di ritorno e gli stessi errori di <code>mkstemp</code> con lo stesso significato, tranne <code>EINVAL</code> che viene restituito se <code>template</code> non è di lunghezza pari ad almeno <code>6+suffixlen</code> ed i 6 caratteri prima del suffisso non sono <code>XXXXXX</code>.</p>	

Le due funzioni, un'estensione non standard fornita dalla *glibc*, sono identiche a `mkstemp` e `mkostemp`, ma consentono di avere un nome del file nella forma `prefissoXXXXXXsuffisso` dove la lunghezza del suffisso deve essere indicata con `suffixlen`.

Infine con OpenBSD è stata introdotta un'altra funzione simile alle precedenti, `mkdtemp`, che crea invece una directory temporanea;⁶³ il suo prototipo è:

<pre>#include <stdlib.h> char *mkdtemp(char *template)</pre>	<p>Crea una directory temporanea.</p>
<p>La funzione ritorna il puntatore al nome della directory in caso di successo e <code>NULL</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EINVAL</code> <code>template</code> non termina con <code>XXXXXX</code>.</p> <p>più gli altri eventuali codici di errore di <code>mkdir</code>.</p>	

La funzione crea una directory temporanea il cui nome è ottenuto sostituendo le `XXXXXX` finali di `template` con permessi `0700` (si veda sez. 4.4.1 per i dettagli). Dato che la creazione della directory è sempre atomica i precedenti problemi di *race condition* non si pongono.

4.3 La manipolazione delle caratteristiche dei file

Come spiegato in sez. 4.1.2 tutte le informazioni generali relative alle caratteristiche di ciascun file, a partire dalle informazioni relative al controllo di accesso, sono mantenute nell'*inode*. Vedremo in questa sezione come sia possibile leggere tutte queste informazioni usando la funzione `stat`, che permette l'accesso a tutti i dati memorizzati nell'*inode*; esamineremo poi le varie funzioni usate per manipolare tutte queste informazioni, eccetto quelle che riguardano la gestione del controllo di accesso, trattate in sez. 4.4.

4.3.1 La lettura delle caratteristiche dei file

La lettura delle informazioni relative ai file è fatta attraverso la famiglia delle funzioni `stat` che sono quelle che usa il comando `ls` per poter ottenere e mostrare tutti i dati relativi ad un file; ne fanno parte le funzioni di sistema `stat`, `fstat` e `lstat`, i cui prototipi sono:

⁶²si tenga presente che `mkostemp` utilizza già `O_CREAT`, `O_EXCL` e `O_RDWR`, che non è il caso di riindicare, dato che ciò potrebbe portare ad errori in altri sistemi operativi.

⁶³la funzione è stata introdotta nella *glibc* a partire dalla versione 2.1.91 ed inserita nello standard POSIX.1-2008.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf)
int lstat(const char *file_name, struct stat *buf)
int fstat(int filedes, struct stat *buf)
```

Leggono le informazioni di un file.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`E_OVERFLOW` il file ha una dimensione che non può essere rappresentata nel tipo `off_t` (può avvenire solo in caso di programmi compilati su piattaforme a 32 bit senza le estensioni (-D `_FILE_OFFSET_BITS=64`) per file a 64 bit).

ed inoltre `EFAULT` ed `ENOMEM`, per `stat` e `lstat` anche `EACCES`, `ELOOP`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR`, per `fstat` anche `EBADF`, nel loro significato generico.

La funzione `stat` legge le informazioni del file indicato da `file_name` e le inserisce nel buffer puntato dall'argomento `buf`; la funzione `lstat` è identica a `stat` eccetto che se `file_name` è un collegamento simbolico vengono lette le informazioni relative ad esso e non al file a cui fa riferimento. Infine `fstat` esegue la stessa operazione su un file già aperto, specificato tramite il suo file descriptor `filedes`.

La struttura `stat` usata da queste funzioni è definita nell'header `sys/stat.h` e in generale dipende dall'implementazione; la versione usata da Linux è mostrata in fig. 4.13, così come riportata dalla pagina di manuale di `stat`. In realtà la definizione effettivamente usata nel kernel dipende dall'architettura e ha altri campi riservati per estensioni come tempi dei file più precisi (vedi sez. 4.3.4).

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t      st_mode;    /* protection */
    nlink_t     st_nlink;   /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device type (if inode device) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t   st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t    st_blocks;  /* number of blocks allocated */
    time_t      st_atime;   /* time of last access */
    time_t      st_mtime;   /* time of last modification */
    time_t      st_ctime;   /* time of last status change */
};
```

Figura 4.13: La struttura `stat` per la lettura delle informazioni dei file.

Si noti come i vari membri della struttura siano specificati come tipi primitivi del sistema, di quelli definiti in tab. 1.2, e dichiarati in `sys/types.h`, con l'eccezione di `blksize_t` e `blkcnt_t` che sono nuovi tipi introdotti per rendersi indipendenti dalla piattaforma.

Benché la descrizione dei commenti di fig. 4.13 sia abbastanza chiara, vale la pena illustrare maggiormente il significato dei campi di `stat` su cui non torneremo in maggior dettaglio nel resto di questa sezione:

- Il campo `st_nlink` contiene il numero di *hard link* che fanno riferimento al file (il cosiddetto *link count*) di cui abbiamo già parlato in numerose occasioni.

- Il campo `st_ino` contiene il numero di *inode* del file, quello viene usato all'interno del filesystem per identificarlo e che può essere usato da un programma per determinare se due *pathname* fanno riferimento allo stesso file.
- Il campo `st_dev` contiene il numero del dispositivo su cui risiede il file (o meglio il suo filesystem). Si tratta dello stesso numero che si usa con `mknod` e che può essere decomposto in *major number* e *minor number* con le macro `major` e `minor` viste in sez. 4.2.5.
- Il campo `st_rdev` contiene il numero di dispositivo associato al file stesso ed ovviamente ha un valore significativo soltanto quando il file è un dispositivo a caratteri o a blocchi.
- Il campo `st_blksize` contiene la dimensione dei blocchi di dati usati nell'I/O su disco, che è anche la dimensione usata per la bufferizzazione dei dati dalle librerie del C per l'interfaccia degli *stream*. Leggere o scrivere blocchi di dati in dimensioni inferiori a questo valore è inefficiente in quanto le operazioni su disco usano comunque trasferimenti di questa dimensione.

Nell'evoluzione del kernel la *system call* che fornisce `stat` è stata modificata più volte per tener conto dei cambiamenti fatti alla struttura `stat`,⁶⁴ in particolare a riguardo ai tempi dei file, di cui è stata aumentata la precisione (torneremo su questo in sez. 4.3.4) ma anche per gli aggiornamenti fatti ai campi `st_ino`, `st_uid` e `st_gid`.

Sulle piattaforme a 32 bit questi cambiamenti, che han visto un aumento delle dimensioni dei campi della struttura per adattarli alle nuove esigenze, sono mascherati dalla *glibc* che attraverso `stat` invoca la versione più recente della *system call* e rimpacchetta i dati se questo è necessario per eseguire dei vecchi programmi. Nelle piattaforme a 64 bit invece è presente un'unica versione della *system call* e la struttura `stat` ha campi di dimensione sufficiente.

Infine a partire dal kernel 2.6.16 è stata introdotta una ulteriore funzione della famiglia, `fstatat` che consente di trattare con sicurezza i *pathname* relativi, la tratteremo in sez. 5.2.4, insieme alla nuova *system call* `statx`, introdotta dal kernel 4.11 per estendere l'interfaccia di `stat` e le informazioni che essa può restituire.

4.3.2 I tipi di file

Abbiamo sottolineato fin dall'introduzione che Linux, come ogni sistema unix-like, supporta oltre ai file ordinari e alle directory una serie di altri “*tipi*” di file che possono stare su un filesystem (elencati in tab. 1.1). Il tipo di file viene ritornato dalle funzioni della famiglia `stat` all'interno del campo `st_mode` di una struttura `stat`.

Macro	Tipo del file
<code>S_ISREG(m)</code>	File normale.
<code>S_ISDIR(m)</code>	Directory.
<code>S_ISCHR(m)</code>	Dispositivo a caratteri.
<code>S_ISBLK(m)</code>	Dispositivo a blocchi.
<code>S_ISFIFO(m)</code>	<i>Fifo</i> .
<code>S_ISLNK(m)</code>	Collegamento simbolico.
<code>S_ISSOCK(m)</code>	Socket.

Tabella 4.6: Macro per i tipi di file (definite in `sys/stat.h`).

Il campo `st_mode` è una maschera binaria in cui l'informazione viene suddivisa nei vari bit che compongono, ed oltre a quelle sul tipo di file, contiene anche le informazioni relative ai permessi su cui torneremo in sez. 4.4.1. Dato che i valori numerici usati per definire il tipo di

⁶⁴questo ha significato l'utilizzo a basso livello di diverse *system call* e diverse versioni della struttura.

file possono variare a seconda delle implementazioni, lo standard POSIX definisce un insieme di macro che consentono di verificare il tipo di file in maniera standardizzata.

Queste macro vengono usate anche da Linux che supporta pure le estensioni allo standard per i collegamenti simbolici e i socket definite da BSD.⁶⁵ L'elenco completo delle macro con cui è possibile estrarre da `st_mode` l'informazione relativa al tipo di file è riportato in tab. 4.6, tutte le macro restituiscono un valore intero da usare come valore logico e prendono come argomento il valore di `st_mode`.

Flag	Valore	Significato
S_IFMT	0170000	Maschera per i bit del tipo di file.
S_IFSOCK	0140000	Socket.
S_IFLNK	0120000	Collegamento simbolico.
S_IFREG	0100000	File regolare.
S_IFBLK	0060000	Dispositivo a blocchi.
S_IFDIR	0040000	Directory.
S_IFCHR	0020000	Dispositivo a caratteri.
S_IFIFO	0010000	<i>Fifo</i> .
S_ISUID	0004000	Set user ID (<i>suid</i>) bit, vedi sez. 4.4.2).
S_ISGID	0002000	Set group ID (<i>sgid</i>) bit, vedi sez. 4.4.2).
S_ISVTX	0001000	<i>Sticky</i> bit, vedi sez. 4.4.2).
S_IRWXU	00700	Maschera per i permessi del proprietario.
S_IRUSR	00400	Il proprietario ha permesso di lettura.
S_IWUSR	00200	Il proprietario ha permesso di scrittura.
S_IXUSR	00100	Il proprietario ha permesso di esecuzione.
S_IRWXG	00070	Maschera per i permessi del gruppo.
S_IRGRP	00040	Il gruppo ha permesso di lettura.
S_IWGRP	00020	Il gruppo ha permesso di scrittura.
S_IXGRP	00010	Il gruppo ha permesso di esecuzione.
S_IRWXO	00007	Maschera per i permessi di tutti gli altri
S_IROTH	00004	Gli altri hanno permesso di lettura.
S_IWOTH	00002	Gli altri hanno permesso di esecuzione.
S_IXOTH	00001	Gli altri hanno permesso di esecuzione.

Tabella 4.7: Costanti per l'identificazione dei vari bit che compongono il campo `st_mode` (definite in `sys/stat.h`).

Oltre alle macro di tab. 4.6, che semplificano l'operazione, è possibile usare direttamente il valore di `st_mode` per ricavare il tipo di file controllando direttamente i vari bit in esso memorizzati. Per questo sempre in `sys/stat.h` sono definite le varie costanti numeriche riportate in tab. 4.7, che definiscono le maschere che consentono di selezionare non solo i dati relativi al tipo di file, ma anche le informazioni relative ai permessi su cui torneremo in sez. 4.4, ed identificare i rispettivi valori.

Le costanti che servono per la identificazione del tipo di file sono riportate nella prima sezione di tab. 4.7, mentre le sezioni successive attengono alle costanti usate per i permessi. Il primo valore dell'elenco è la maschera binaria `S_IFMT` che permette di estrarre da `st_mode` (con un AND aritmetico) il blocco di bit nei quali viene memorizzato il tipo di file. I valori successivi sono le costanti corrispondenti ai vari tipi di file, e possono essere usate per verificare la presenza del tipo di file voluto ed anche, con opportune combinazioni, alternative fra più tipi di file.

Si tenga presente però che a differenza dei permessi, l'informazione relativa al tipo di file non è una maschera binaria, per questo motivo se si volesse impostare una condizione che

⁶⁵Le ultime due macro di tab. 4.6, che non sono presenti nello standard POSIX fino alla revisione POSIX.1-1996.

permetta di controllare se un file è una directory o un file ordinario non si possono controllare dei singoli bit, ma si dovrebbe usare una macro di preprocessore come:

```
#define IS_RGDR(x) (((x)&S_IFMT&S_IFDIR) | ((x)&S_IFMT&S_IFREG))
```

in cui si estraggono ogni volta da `st_mode` i bit relativi al tipo di file e poi si effettua il confronto con i due possibili tipi di file.

4.3.3 Le dimensioni dei file

Abbiamo visto in fig. 4.13 che campo `st_size` di una struttura `stat` contiene la dimensione del file in byte. In realtà questo è vero solo se si tratta di un file regolare contenente dei dati; nel caso di un collegamento simbolico invece la dimensione è quella del *pathname* che il collegamento stesso contiene, e per una directory quella dello spazio occupato per le voci della stessa (che dipende da come queste vengono mantenute dal filesystem), infine per le *fifo*, i socket ed i file di dispositivo questo campo è sempre nullo.

Il campo `st_blocks` invece definisce la lunghezza del file espressa in numero di blocchi di 512 byte. La differenza con `st_size` è che in questo caso si fa riferimento alla quantità di spazio disco allocata per il file, e non alla dimensione dello stesso che si otterrebbe leggendolo sequenzialmente.

Si deve tener presente infatti che la lunghezza del file riportata in `st_size` non è detto che corrisponda all'occupazione dello spazio su disco, e non solo perché la parte finale del file potrebbe riempire parzialmente un blocco. In un sistema unix-like infatti è possibile l'esistenza dei cosiddetti *sparse file*, cioè file in cui sono presenti dei “*buchi*” (*holes* nella nomenclatura inglese) che si formano tutte le volte che si va a scrivere su un file dopo aver eseguito uno spostamento oltre la sua fine (tratteremo in dettaglio l'argomento in sez. 5.1.3).

In questo caso si avranno risultati differenti a seconda del modo in cui si calcola la lunghezza del file, ad esempio il comando `du`, (che riporta il numero di blocchi occupati) potrà dare una dimensione inferiore, mentre se si legge il contenuto del file (ad esempio usando il comando `wc -c`), dato che in tal caso per i “*buchi*” vengono restituiti degli zeri, si avrà lo stesso risultato di `ls`.

Se è sempre possibile allargare un file, scrivendoci sopra o usando la funzione `lseek` (vedi sez. 5.1.3) per spostarsi oltre la sua fine, esistono anche casi in cui si può avere bisogno di effettuare un troncamento, scartando i dati presenti al di là della dimensione scelta come nuova fine del file.

Un file può sempre essere troncato a zero aprendolo con il flag `O_TRUNC` (vedi sez. 5.1.2), ma questo è un caso particolare; per qualunque altra dimensione si possono usare le due funzioni di sistema `truncate` e `ftruncate`, i cui prototipi sono:

```
#include <unistd.h>
int ftruncate(int fd, off_t length)
int truncate(const char *file_name, off_t length)
```

Troncano un file.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINTR` si è stati interrotti da un segnale.

`EINVAL` `length` è negativa o maggiore delle dimensioni massime di un file.

`EPERM` il filesystem non supporta il troncamento di un file.

`ETXTBSY` il file è un programma in esecuzione.

per entrambe, mentre per `ftruncate` si avranno anche:

`EBADF` `fd` non è un file descriptor.

`EINVAL` `fd` non è un riferimento a un file o non è aperto in scrittura.

e per `truncate` si avranno anche:

`EACCES` non si ha il permesso di scrittura sul file o il permesso di attraversamento di una delle componenti del *pathname*.

`EISDIR` `file_name` fa riferimento ad una directory.

ed inoltre `EFAULT`, `EIO`, `ELOOP`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR` e `EROFS` nel loro significato generico.

Entrambe le funzioni fan sì che la dimensione del file sia troncata ad un valore massimo specificato da `length`, e si distinguono solo per il fatto che il file viene indicato con un *pathname* per `truncate` e con un file descriptor per `ftruncate`. Si tenga presente che se il file è più lungo della lunghezza specificata i dati in eccesso saranno perduti.

Il comportamento in caso di lunghezza del file inferiore a `length` non è specificato e dipende dall'implementazione: il file può essere lasciato invariato o esteso fino alla lunghezza scelta. Nel caso di Linux viene esteso con la creazione di un *bucu* nel file e ad una lettura si otterranno degli zeri, si tenga presente però che questo comportamento è supportato solo per filesystem nativi, ad esempio su un filesystem non nativo come il VFAT di Windows questo non è possibile.

4.3.4 I tempi dei file

Il sistema mantiene per ciascun file tre tempi, che sono registrati nell'*inode* insieme agli altri attributi del file. Questi possono essere letti tramite la funzione `stat`, che li restituisce attraverso tre campi della struttura `stat` di fig. 4.13. Il significato di questi tempi e dei relativi campi della struttura `stat` è illustrato nello schema di tab. 4.8, dove è anche riportato un esempio delle funzioni che effettuano cambiamenti su di essi. Il valore del tempo è espresso nel cosiddetto *calendar time*, su cui torneremo in dettaglio in sez. 6.4.

Membro	Significato	Funzione	Opzione di <code>ls</code>
<code>st_atime</code>	ultimo accesso ai dati del file	<code>read</code> , <code>utime</code>	<code>-u</code>
<code>st_mtime</code>	ultima modifica ai dati del file	<code>write</code> , <code>utime</code>	<code>default</code>
<code>st_ctime</code>	ultima modifica ai dati dell' <i>inode</i>	<code>chmod</code> , <code>utime</code>	<code>-c</code>

Tabella 4.8: I tre tempi associati a ciascun file.

Il primo punto da tenere presente è la differenza fra il cosiddetto tempo di ultima modifica (il *modification time*) riportato in `st_mtime`, ed il tempo di ultimo cambiamento di stato (il *change status time*) riportato in `st_ctime`. Il primo infatti fa riferimento ad una modifica del contenuto di un file, mentre il secondo ad una modifica dei metadati dell'*inode*. Dato che esistono molte operazioni, come la funzione `link` e altre che vedremo in seguito, che modificano solo le informazioni contenute nell'*inode* senza toccare il contenuto del file, diventa necessario l'utilizzo di questo secondo tempo.

Il tempo di ultima modifica viene usato ad esempio da programmi come `make` per decidere quali file necessitano di essere ricompilati perché più recenti dei loro sorgenti oppure dai programmi di backup, talvolta insieme anche al tempo di cambiamento di stato, per decidere quali file devono essere aggiornati nell'archiviazione. Il tempo di ultimo accesso viene di solito usato per identificare i file che non vengono più utilizzati per un certo lasso di tempo. Ad esempio un programma come `leafnode` lo usa per cancellare gli articoli letti più vecchi, mentre `mutt` lo usa per marcare i messaggi di posta che risultano letti.

Il sistema non tiene mai conto dell'ultimo accesso all'*inode*, pertanto funzioni come `access` o `stat` non hanno alcuna influenza sui tre tempi. Il comando `ls` (quando usato con le opzioni `-l` o `-t`) mostra i tempi dei file secondo lo schema riportato nell'ultima colonna di tab. 4.8. Si noti anche come in tab. 4.8 non venga riportato il *tempo di creazione* di un file; in un sistema unix-like infatti questo non esiste, e non è previsto dall'interfaccia classica, ma essendo usato da altri sistemi operativi (in particolare Windows) in tutti i filesystem più recenti ne viene supportata la registrazione, ed a partire dal kernel 4.11 è diventato possibile anche ottenerne la lettura con la nuova *system call* `statx` (che tratteremo in sez. 5.2.4).

L'aggiornamento del tempo di ultimo accesso è stato a lungo considerato un difetto progettuale di Unix, questo infatti comporta la necessità di effettuare un accesso in scrittura sul disco anche in tutti i casi in cui questa informazione non interessa e sarebbe possibile avere un semplice accesso in lettura sui dati bufferizzati. Questo comporta un ovvio costo sia in termini di prestazioni, che di consumo di risorse come la batteria per i portatili, o i cicli di riscrittura per i dischi su memorie riscrivibili.

Per questo motivo abbiamo visto in sez. 4.1.4 come nel corso dello sviluppo del kernel siano stati introdotti degli opportuni *mount flag* che consentono di evitare di aggiornare continuamente una informazione che nella maggior parte dei casi non ha un interesse rilevante. Per questo motivo i valori dell'*access time* possono dipendere dalle opzioni di montaggio, ed anche, essendo stato cambiato il comportamento di default a partire dalla versione 2.6.30, dal kernel che si sta usando.

In generale quello che avviene con i kernel più recenti è che il tempo di ultimo accesso viene aggiornato solo se è precedente al tempo di ultima modifica o cambiamento, o se è cambiato ed passato più di un giorno dall'ultimo aggiornamento. Così si può rendere evidente che vi è stato un accesso dopo una modifica, e che il file viene comunque osservato a cadenza regolare, conservando le informazioni veramente utili senza consumare inutilmente risorse in continue scritture per mantenere costantemente aggiornata una informazione che a questo punto non ha più nessuna rilevanza pratica.⁶⁶

L'effetto delle varie funzioni di manipolazione dei file sui relativi tempi è illustrato in tab. 4.9, facendo riferimento al comportamento classico per quanto riguarda `st_atime`. Si sono riportati gli effetti sia per il file a cui si fa riferimento, sia per la directory che lo contiene. Questi ultimi possono essere capiti immediatamente se si tiene conto di quanto già detto e ripetuto a partire da sez. 4.1.2, e cioè che anche le directory sono anch'esse file che contengono una lista di nomi, che il sistema tratta in maniera del tutto analoga a tutti gli altri.

Per questo motivo tutte le volte che compiremo un'operazione su un file che comporta una modifica del nome contenuto nella directory, andremo anche a scrivere sulla directory che lo contiene cambiandone il tempo di ultima modifica. Un esempio di questo tipo di operazione può essere la cancellazione di un file, invece leggere o scrivere o cambiare i permessi di un file ha effetti solo sui tempi di quest'ultimo.

Si ricordi infine come `st_ctime` non è il tempo di creazione del file, che in Unix non esiste, anche se può corrispondervi per file che non sono mai stati modificati. Per questo motivo,

⁶⁶qualora ce ne fosse la necessità è comunque possibile, tramite l'opzione di montaggio `strictatime`, richiedere in ogni caso il comportamento tradizionale.

Funzione	File o directory del riferimento			Directory contenente il riferimento			Note
	(a)	(m)	(c)	(a)	(m)	(c)	
chmod, fchmod	—	—	•	—	—	—	con O_CREATE con O_TRUNC
chown, fchown	—	—	•	—	—	—	
creat	•	•	•	—	•	•	
creat	—	•	•	—	•	•	
exec	•	—	—	—	—	—	con O_CREATE con O_TRUNC
lchown	—	—	•	—	—	—	
link	—	—	•	—	•	•	
mkdir	•	•	•	—	•	•	
mknod	•	•	•	—	•	•	con O_CREATE con O_TRUNC
mkfifo	•	•	•	—	•	•	
open	•	•	•	—	•	•	
open	—	•	•	—	—	—	
pipe	•	•	•	—	—	—	se esegue unlink se esegue rmdir per ambo gli argomenti
read	•	—	—	—	—	—	
remove	—	—	•	—	•	•	
remove	—	—	—	—	•	•	
rename	—	—	•	—	•	•	
rmdir	—	—	—	—	•	•	
truncate	—	•	•	—	—	—	
ftruncate	—	•	•	—	—	—	
unlink	—	—	•	—	•	•	
utime	•	•	•	—	—	—	
utimes	•	•	•	—	—	—	
write	—	•	•	—	—	—	

Tabella 4.9: Prospetto dei cambiamenti effettuati sui tempi di ultimo accesso (a), ultima modifica (m) e ultimo cambiamento di stato (c) dalle varie funzioni operanti su file e directory.

a differenza di quanto avviene con altri sistemi operativi, quando si copia un file, a meno di preservare esplicitamente i tempi (ad esempio con l'opzione `-p` di `cp`) esso avrà sempre il tempo corrente in cui si è effettuata la copia come data di ultima modifica.

I tempi di ultimo accesso ed ultima modifica possono essere modificati esplicitamente usando la funzione di sistema `utime`, il cui prototipo è:

<pre>#include <utime.h> int utime(const char *filename, struct utimbuf *times)</pre>		Modifica i tempi di ultimo accesso ed ultima modifica di un file.
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:		
EACCES	non c'è il permesso di attraversamento per uno dei componenti di <code>filename</code> o <code>times</code> è NULL e non si ha il permesso di scrittura sul file, o non si è proprietari del file o non si hanno i privilegi di amministratore.	
EPERM	<code>times</code> non è NULL, e non si è proprietari del file o non si hanno i privilegi di amministratore.	
ed inoltre ENOENT e EROFS nel loro significato generico.		

La funzione cambia i tempi di ultimo accesso e di ultima modifica del file specificato dall'argomento `filename`, e richiede come secondo argomento il puntatore ad una struttura `utimbuf`, la cui definizione è riportata in fig. 4.14, con i nuovi valori di detti tempi (rispettivamente nei campi `actime` e `modtime`). Se si passa un puntatore nullo verrà impostato il tempo corrente.

L'effetto della funzione ed i privilegi necessari per eseguirla dipendono dal valore dell'argomento `times`. Se è NULL la funzione imposta il tempo corrente ed è sufficiente avere accesso in scrittura al file o essere proprietari del file o avere i privilegi di amministratore. Se invece

```

struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};

```

Figura 4.14: La struttura `utimbuf`, usata da `utime` per modificare i tempi dei file.

si è specificato un valore diverso la funzione avrà successo solo se si è proprietari del file o se si hanno i privilegi di amministratore.⁶⁷ In entrambi i casi per verificare la proprietà del file viene utilizzato l'*UID* effettivo del processo.

Si tenga presente che non è possibile modificare manualmente il tempo di cambiamento di stato del file, che viene aggiornato direttamente dal kernel tutte le volte che si modifica l'*inode*, e quindi anche alla chiamata di `utime`. Questo serve anche come misura di sicurezza per evitare che si possa modificare un file nascondendo completamente le proprie tracce. In realtà la cosa resta possibile se si è in grado di accedere al file di dispositivo, scrivendo direttamente sul disco senza passare attraverso il filesystem, ma ovviamente in questo modo la cosa è più complicata da realizzare.⁶⁸

A partire dal kernel 2.6 la risoluzione dei tempi dei file, che nei campi di tab. 4.8 è espressa in secondi, è stata portata ai nanosecondi per la gran parte dei filesystem. L'ulteriore informazione può essere ottenuta attraverso altri campi appositamente aggiunti alla struttura `stat`. Se si sono definite le macro `_BSD_SOURCE` o `_SVID_SOURCE` questi sono `st_atim.tv_nsec`, `st_mtim.tv_nsec` e `st_ctim.tv_nsec` se queste non sono definite, `st_atimensec`, `st_mtimensec` e `st_ctimensec`. Qualora il supporto per questa maggior precisione sia assente questi campi aggiuntivi saranno nulli.

Per la gestione di questi nuovi valori è stata definita una seconda funzione di sistema, `utimes`, che consente di specificare tempi con maggior precisione; il suo prototipo è:

```
#include <sys/time.h>
int utimes(const char *filename, struct timeval times[2])
    Modifica i tempi di ultimo accesso e ultima modifica di un file.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà gli stessi valori di `utime`.

La funzione è del tutto analoga alla precedente `utime` ma usa come secondo argomento un vettore di due strutture `timeval`, la cui definizione è riportata in fig. 4.15, che consentono di indicare i tempi con una precisione del microsecondo. Il primo elemento di `times` indica il valore per il tempo di ultimo accesso, il secondo quello per il tempo di ultima modifica. Se si indica come secondo argomento un puntatore nullo di nuovo verrà utilizzato il tempo corrente.

Oltre ad `utimes` su Linux sono presenti altre due funzioni,⁶⁹ `futimes` e `lutimes`, che consentono rispettivamente di effettuare la modifica utilizzando un file già aperto o di eseguirla direttamente su un collegamento simbolico. I relativi prototipi sono:

⁶⁷per essere precisi la capacità `CAP_FOWNER`, vedi sez. 9.1.1.

⁶⁸esistono comunque molti programmi che consentono di farlo con relativa semplicità per cui non si dia per scontato che il valore sia credibile in caso di macchina compromessa.

⁶⁹le due funzioni non sono definite in nessuno standard, ma sono presenti, oltre che su Linux, anche su BSD; sono accessibili definendo `_DEFAULT_SOURCE` dalla *glibc* 2.19 o `_GNU_SOURCE` prima.

```

struct timeval
{
    time_t tv_sec;           /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};

```

Figura 4.15: La struttura `timeval` usata per indicare valori di tempo con la precisione del microsecondo.

```

#include <sys/time.h>
int futimes(int fd, const struct timeval tv[2])
    Cambia i tempi di un file già aperto.
int lutimes(const char *filename, const struct timeval tv[2])
    Cambia i tempi di un collegamento simbolico.

```

Le funzioni ritornano 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno gli stessi valori di `utimes`, con in più per `futimes`:

EBADF `fd` non è un file descriptor.

ENOSYS il filesystem `/proc` non è accessibile per `futimes` o la funzione non è supportata per `lutimes`.

Le due funzioni hanno lo stesso comportamento di `utimes` e richiedono gli stessi privilegi per poter operare, la differenza è che con `futimes` si può indicare il file su cui operare se questo è già aperto facendo riferimento al suo file descriptor, mentre con `lutimes` nel caso in cui `filename` sia un collegamento simbolico saranno modificati i suoi tempi invece di quelli del file a cui esso punta.

Nonostante il kernel nelle versioni più recenti supporti, come accennato, risoluzioni dei tempi dei file fino al nanosecondo, le funzioni fin qui esaminate non consentono di impostare valori con questa precisione. Per questo sono state introdotte due nuove funzioni di sistema, `utimensat` (che vedremo in sez. 5.2.4 insieme alle altre *at-functions*), e `futimens`, il cui prototipo è:

```

#include <sys/time.h>
int futimens(int fd, const struct timespec times[2])
    Cambia i tempi di un file già aperto.

```

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES si è richiesta l'impostazione del tempo corrente ma non si ha il permesso di scrittura sul file, o non si è proprietari del file o non si hanno i privilegi di amministratore; oppure il file è immutabile (vedi sez. 4.4.1).

EBADF `fd` non è un file descriptor valido.

EFAULT `times` non è un puntatore valido.

EINVAL si sono usati dei valori non corretti per i tempi di `times`.

EPERM si è richiesto un cambiamento nei tempi non al tempo corrente, ma non si è proprietari del file o non si hanno i privilegi di amministratore; oppure il file è immutabile o *append-only* (vedi sez. 4.4.1).

ed inoltre per entrambe **EROFS** nel suo significato generico.

La funzione è sostanzialmente una estensione di `futimes` che consente di specificare i tempi con precisione maggiore. Per questo per indicare i valori dei tempi da impostare utilizza un vettore `times` di due strutture `timespec` che permettono di indicare il valore del tempo con una precisione fino al nanosecondo (se ne è riportata la definizione in fig. 4.16).

```

struct timespec {
    time_t    tv_sec;           /* seconds */
    long int  tv_nsec;         /* nanoseconds */
};

```

Figura 4.16: La struttura `timespec` usata per indicare valori di tempo con la precisione del nanosecondo.

Come per le precedenti funzioni il primo elemento di `times` indica il tempo di ultimo accesso ed il secondo quello di ultima modifica, e se si usa il valore `NULL` verrà impostato il tempo corrente sia per l'ultimo accesso che per l'ultima modifica.

Nei singoli elementi di `times` si possono inoltre utilizzare due valori speciali per il campo `tv_nsec`: con `UTIME_NOW` si richiede l'uso del tempo corrente, mentre con `UTIME_OMIT` si richiede di non impostare il tempo. Si può così aggiornare in maniera specifica soltanto uno fra il tempo di ultimo accesso e quello di ultima modifica. Quando si usa uno di questi valori speciali per `tv_nsec` il corrispondente valore di `tv_sec` viene ignorato.

Questa funzione e `utimensat` sono una estensione definita nella revisione POSIX.1-2008 dello standard POSIX; in Linux sono state introdotte a partire dal kernel 2.6.22, e sono supportate dalla *glibc* a partire dalla versione 2.6, si tenga presente però che per kernel precedenti il 2.6.26 le due funzioni sono difettose nel rispetto di alcuni requisiti minori dello standard e nel controllo della correttezza dei tempi, per i dettagli dei quali si rimanda alla pagina di manuale.

4.4 Il controllo di accesso ai file

Una delle caratteristiche fondamentali di tutti i sistemi unix-like è quella del controllo di accesso ai file, che viene implementato per qualunque filesystem standard.⁷⁰ In questa sezione ne esamineremo i concetti essenziali e le funzioni usate per gestirne i vari aspetti.

4.4.1 I permessi per l'accesso ai file

Ad ogni file Linux associa sempre, oltre ad un insieme di permessi, l'utente che ne è proprietario (il cosiddetto *owner*) ed un gruppo di appartenenza, indicati dagli identificatori di utente e gruppo (*UID* e *GID*) di cui abbiamo già parlato in sez. 3.2.1.⁷¹ Anche questi sono mantenuti sull'*inode* insieme alle altre proprietà e sono accessibili da programma tramite la funzione `stat` (trattata in sez. 4.3.1), che restituisce l'utente proprietario nel campo `st_uid` ed il gruppo proprietario nel campo `st_gid` della omonima struttura `stat`.

Il controllo di accesso ai file segue un modello abbastanza semplice che prevede tre permessi fondamentali strutturati su tre livelli di accesso. Esistono varie estensioni a questo modello,⁷²

⁷⁰per filesystem standard si intende un filesystem che implementi le caratteristiche previste dallo standard POSIX; in Linux sono utilizzabili anche filesystem di altri sistemi operativi, che non supportano queste caratteristiche.

⁷¹questo è vero solo per filesystem di tipo Unix, ad esempio non è vero per il filesystem VFAT di Windows, che non fornisce nessun supporto per l'accesso multiutente, e per il quale queste proprietà vengono assegnate in maniera fissa con opportune opzioni di montaggio.

⁷²come le *Access Control List* che sono state aggiunte ai filesystem standard con opportune estensioni (vedi sez. 4.5.2) per arrivare a meccanismi di controllo ancora più sofisticati come il *Mandatory Access Control* di *SELinux* e delle altre estensioni come *Smack* o *AppArmor*.

ma nella maggior parte dei casi il meccanismo standard è più che sufficiente a soddisfare tutte le necessità più comuni. I tre permessi di base associati ad ogni file sono:

- il permesso di lettura (indicato con la lettera *r*, dall'inglese *read*).
- il permesso di scrittura (indicato con la lettera *w*, dall'inglese *write*).
- il permesso di esecuzione (indicato con la lettera *x*, dall'inglese *execute*).

mentre i tre livelli su cui sono divisi i privilegi sono:

- i privilegi per l'utente proprietario del file.
- i privilegi per un qualunque utente faccia parte del gruppo cui appartiene il file.
- i privilegi per tutti gli altri utenti.

L'insieme dei permessi viene espresso con un numero a 12 bit; di questi i nove meno significativi sono usati a gruppi di tre per indicare i permessi base di lettura, scrittura ed esecuzione e sono applicati rispettivamente al proprietario, al gruppo, a tutti gli altri.

Figura 4.17: Lo schema dei bit utilizzati per specificare i permessi di un file contenuti nel campo `st_mode` di `stat`.

I restanti tre bit (noti come *suid bit*, *sgid bit*, e *sticky bit*) sono usati per indicare alcune caratteristiche più complesse del meccanismo del controllo di accesso su cui torneremo in seguito (in sez. 4.4.2), lo schema di allocazione dei bit è riportato in fig. 4.17. Come tutte le altre proprietà di un file anche i permessi sono memorizzati nell'*inode*, e come accennato in sez. 4.3.2 essi vengono restituiti in una parte del campo `st_mode` della struttura `stat` (si veda di nuovo fig. 4.13).

<code>st_mode</code> bit	Significato
S_IRUSR	<i>user-read</i> , l'utente può leggere.
S_IWUSR	<i>user-write</i> , l'utente può scrivere.
S_IXUSR	<i>user-execute</i> , l'utente può eseguire.
S_IRGRP	<i>group-read</i> , il gruppo può leggere.
S_IWGRP	<i>group-write</i> , il gruppo può scrivere.
S_IXGRP	<i>group-execute</i> , il gruppo può eseguire.
S_IROTH	<i>other-read</i> , tutti possono leggere.
S_IWOTH	<i>other-write</i> , tutti possono scrivere.
S_IXOTH	<i>other-execute</i> , tutti possono eseguire.

Tabella 4.10: I bit dei permessi di accesso ai file, come definiti in `<sys/stat.h>`

In genere ci si riferisce ai tre livelli dei privilegi usando le lettere *u* (per *user*), *g* (per *group*) e *o* (per *other*), inoltre se si vuole indicare tutti i raggruppamenti insieme si usa la lettera *a* (per *all*). Si tenga ben presente questa distinzione dato che in certi casi, mutuando la terminologia in uso a suo tempo nel VMS, si parla dei permessi base come di permessi per *owner*, *group* ed *all*, le cui iniziali possono dar luogo a confusione. Le costanti che permettono di accedere al valore numerico di questi bit nel campo `st_mode`, già viste in tab. 4.7, sono riportate per chiarezza una seconda volta in tab. 4.10.

I permessi vengono usati in maniera diversa dalle varie funzioni, e a seconda che si riferiscano a dei file, dei collegamenti simbolici o delle directory; qui ci limiteremo ad un riassunto delle regole generali, entrando nei dettagli più avanti.

La prima regola è che per poter accedere ad un file attraverso il suo *pathname* occorre il permesso di esecuzione in ciascuna delle directory che compongono il *pathname*; lo stesso vale

per aprire un file nella directory corrente (per la quale appunto serve il diritto di esecuzione). Per una directory infatti il permesso di esecuzione significa che essa può essere attraversata nella risoluzione del *pathname*, e per questo viene anche chiamato permesso di attraversamento. Esso è sempre distinto dal permesso di lettura che invece implica che si può leggere il contenuto della directory.

Questo significa che se si ha il permesso di esecuzione senza permesso di lettura si potrà lo stesso aprire un file all'interno di una directory (se si hanno i permessi adeguati per il medesimo) ma non si potrà vederlo con `ls` mancando il permesso di leggere il contenuto della directory. Per crearlo o rinominarlo o cancellarlo invece occorrerà avere anche il permesso di scrittura per la directory.

Avere il permesso di lettura per un file consente di aprirlo con le opzioni (si veda quanto riportato in sez. 5.1.2) di sola lettura o di lettura/scrittura e leggerne il contenuto. Avere il permesso di scrittura consente di aprire un file in sola scrittura o lettura/scrittura e modificarne il contenuto, lo stesso permesso è necessario per poter troncare il file o per aggiornare il suo tempo di ultima modifica al tempo corrente, ma non per modificare arbitrariamente quest'ultimo, operazione per la quale, come per buona parte delle modifiche effettuate sui metadati del file, occorre esserne i proprietari.

Non si può creare un file fintanto che non si disponga del permesso di esecuzione e di quello di scrittura per la directory di destinazione. Gli stessi permessi occorrono per cancellare un file da una directory (si ricordi che questo non implica necessariamente la rimozione del contenuto del file dal disco).

Per la cancellazione non è necessario nessun tipo di permesso per il file stesso dato che, come illustrato in sez. 4.2.1 esso non viene toccato, nella cancellazione infatti viene solo modificato il contenuto della directory, rimuovendo la voce che ad esso fa riferimento. Lo stesso vale per poter rinominare o spostare il file in altra directory, in entrambi i casi occorrerà il permesso di scrittura sulle directory che si vanno a modificare.

Per poter eseguire un file, che sia un programma compilato od uno script di shell, od un altro tipo di file eseguibile riconosciuto dal kernel, occorre oltre al permesso di lettura per accedere al contenuto avere anche il permesso di esecuzione. Inoltre solo i file regolari possono essere eseguiti. Per i file di dispositivo i permessi validi sono solo quelli di lettura e scrittura, che corrispondono al poter eseguire dette operazioni sulla periferica sottostante.

I permessi per un collegamento simbolico sono ignorati, contano quelli del file a cui fa riferimento; per questo in genere il comando `ls` riporta per un collegamento simbolico tutti i permessi come concessi. Utente e gruppo a cui esso appartiene vengono pure ignorati quando il collegamento viene risolto, vengono controllati solo quando viene richiesta la rimozione del collegamento e quest'ultimo è in una directory con lo *sticky bit* impostato (si veda sez. 4.4.2).

La procedura con cui il kernel stabilisce se un processo possiede un certo permesso (di lettura, scrittura o esecuzione) si basa sul confronto fra l'utente e il gruppo a cui il file appartiene (i valori di `st_uid` e `st_gid` accennati in precedenza) e l'*UID* effettivo, il *GID* effettivo e gli eventuali *GID* supplementari del processo.⁷³

Per una spiegazione dettagliata degli identificatori associati ai processi si veda sez. 3.2; normalmente, a parte quanto vedremo in sez. 4.4.2, l'*UID* effettivo e il *GID* effettivo corrispondono ai valori dell'*UID* e del *GID* dell'utente che ha lanciato il processo, mentre i *GID* supplementari sono quelli dei gruppi cui l'utente appartiene.

I passi attraverso i quali viene stabilito se il processo possiede il diritto di accesso sono i seguenti:

⁷³in realtà Linux, per quanto riguarda l'accesso ai file, utilizza gli identificatori del gruppo *filesystem* (si ricordi quanto esposto in sez. 3.2), ma essendo questi del tutto equivalenti ai primi, eccetto il caso in cui si voglia scrivere un server NFS, ignoreremo questa differenza.

1. Se l'*UID* effettivo del processo è zero (corrispondente all'amministratore) l'accesso è sempre garantito senza nessun controllo. Per questo motivo l'amministratore ha piena libertà di accesso a tutti i file.
2. Se l'*UID* effettivo del processo è uguale all'*UID* del proprietario del file (nel qual caso si dice che il processo è proprietario del file) allora:
 - se il relativo⁷⁴ bit dei permessi d'accesso dell'utente è impostato, l'accesso è consentito;
 - altrimenti l'accesso è negato.
3. Se il *GID* effettivo del processo o uno dei *GID* supplementari del processo corrispondono al *GID* del file allora:
 - se il bit dei permessi d'accesso del gruppo è impostato, l'accesso è consentito;
 - altrimenti l'accesso è negato.
4. Se il bit dei permessi d'accesso per tutti gli altri è impostato, l'accesso è consentito, altrimenti l'accesso è negato.

Si tenga presente che questi passi vengono eseguiti esattamente in quest'ordine. Questo vuol dire che se un processo è il proprietario di un file, l'accesso è consentito o negato solo sulla base dei permessi per l'utente; i permessi per il gruppo non vengono neanche controllati. Lo stesso vale se il processo appartiene ad un gruppo appropriato, in questo caso i permessi per tutti gli altri non vengono controllati.

Questo significa che se si è proprietari di un file ma non si ha il permesso di scrittura, non vi si potrà scrivere anche se questo fosse scrivibile per tutti gli altri. Permessi di questo tipo sono ovviamente poco ortodossi, e comunque, come vedremo in sez. 4.4.3, il proprietario di un file può sempre modificarne i permessi, e riassegnarsi un eventuale permesso di scrittura mancante.

A questi che sono i permessi ordinari si aggiungono, per i filesystem che supportano questa estensione, due permessi speciali mantenuti nei cosiddetti *file attributes*, che si possono leggere ed impostare con i comandi `lsattr` e `chattr`.⁷⁵

Il primo è il cosiddetto attributo di immutabilità (*immutable*, identificato dalla lettera *i*) che impedisce ogni modifica al file, *inode* compreso. Questo significa non solo che non se ne può cambiare il contenuto, ma neanche nessuna delle sue proprietà, ed in particolare non si può modificare nei permessi o nei tempi o nel proprietario ed inoltre, visto che non se può modificare il *link count*, non si può neanche cancellare, rinominare, o creare *hard link* verso di esso.

Il secondo è il cosiddetto attributo di *append-only*, (identificato dalla lettera *a*) che consente soltanto la scrittura in coda al file. Il file cioè può essere soltanto esteso nel contenuto, ma i suoi metadati, a parte i tempi che però possono essere impostati al valore corrente, non possono essere modificati in alcun modo, quindi di nuovo non si potrà cancellare, rinominare, o modificare nei permessi o nelle altre proprietà.

Entrambi questi attributi attivano queste restrizioni a livello di filesystem, per cui a differenza dei permessi ordinari esse varranno per qualunque utente compreso l'amministratore. L'amministratore è l'unico che può attivare o disattivare questi attributi,⁷⁶ e potendo rimuoverli è comunque capace di tornare in grado di eseguire qualunque operazione su un file immutabile o *append-only*.

⁷⁴per relativo si intende il bit di *user-read* se il processo vuole accedere in lettura, quello di *user-write* per l'accesso in scrittura, ecc.

⁷⁵per l'utilizzo di questi comandi e per le spiegazioni riguardo tutti gli altri *file attributes* si rimanda alla sezione 1.4.4 di [?].

⁷⁶più precisamente un processo con la capacità `CAP_LINUX_IMMUTABLE`, vedi sez. 9.1.1.

4.4.2 I bit dei permessi speciali

Come si è accennato (in sez. 4.4.1) nei dodici bit del campo `st_mode` di `stat` che vengono usati per il controllo di accesso oltre ai bit dei permessi veri e propri, ci sono altri tre bit che vengono usati per indicare alcune proprietà speciali dei file. Due di questi sono i bit detti *suid* (da *set-user-ID bit*) e *sgid* (da *set-group-ID bit*) che sono identificati dalle costanti `S_ISUID` e `S_ISGID`.

Come spiegato in dettaglio in sez. 3.1.6, quando si lancia un programma il comportamento normale del kernel è quello di impostare gli identificatori del gruppo *effective* del nuovo processo al valore dei corrispondenti del gruppo *real* del processo corrente, che normalmente corrispondono a quelli dell'utente con cui si è entrati nel sistema.

Se però il file del programma, che ovviamente deve essere eseguibile,⁷⁷ ha il bit *suid* impostato, il kernel assegnerà come *UID* effettivo al nuovo processo l'*UID* del proprietario del file al posto dell'*UID* del processo originario. Avere il bit *sgid* impostato ha lo stesso effetto sul *GID* effettivo del processo. È comunque possibile riconoscere questa situazione perché il cambiamento viene effettuato solo sugli identificativi del gruppo *effective*, mentre quelli dei gruppi *real* e *saved* restano quelli dell'utente che ha eseguito il programma.

I bit *suid* e *sgid* vengono usati per permettere agli utenti normali di usare programmi che richiedono privilegi speciali. L'esempio classico è il comando `passwd` che ha la necessità di modificare il file delle password, quest'ultimo ovviamente può essere scritto solo dall'amministratore, ma non è necessario chiamare l'amministratore per cambiare la propria password. Infatti il comando `passwd` appartiene in genere all'utente *root* ma ha il bit *suid* impostato, per cui quando viene lanciato da un utente normale ottiene comunque i privilegi di amministratore.

Chiaramente avere un processo che ha privilegi superiori a quelli che avrebbe normalmente l'utente che lo ha lanciato comporta vari rischi, e questo tipo di programmi devono essere scritti accuratamente per evitare che possano essere usati per guadagnare privilegi non consentiti (l'argomento è affrontato in dettaglio in sez. 3.2).

La presenza dei bit *suid* e *sgid* su un file può essere rilevata con il comando `ls -l`, che visualizza una lettera "s" al posto della "x" in corrispondenza dei permessi di utente o gruppo. La stessa lettera "s" può essere usata nel comando `chmod` per impostare questi bit. Infine questi bit possono essere controllati all'interno di `st_mode` con l'uso delle due costanti `S_ISUID` e `S_ISGID`, i cui valori sono riportati in tab. 4.7.

Gli stessi bit vengono ad assumere un significato completamente diverso per le directory, normalmente infatti Linux usa la convenzione di SVr4 per indicare con questi bit l'uso della semantica BSD nella creazione di nuovi file (si veda sez. 4.4.4 per una spiegazione dettagliata al proposito).

Infine Linux utilizza il bit *sgid* per un'ulteriore estensione mutuata da SVr4. Il caso in cui un file ha il bit *sgid* impostato senza che lo sia anche il corrispondente bit di esecuzione viene utilizzato per attivare per quel file il *mandatory locking* (affronteremo questo argomento in dettaglio più avanti, in sez. 10.1.5).

L'ultimo dei bit rimanenti, identificato dalla costante `S_ISVTX`, è in parte un rimasuglio delle origini dei sistemi Unix. A quell'epoca infatti la memoria virtuale e l'accesso ai file erano molto meno sofisticati e per ottenere la massima velocità possibile per i programmi usati più comunemente si poteva impostare questo bit.

L'effetto di questo bit era che il segmento di testo del programma (si veda sez. 2.2.2 per i dettagli) veniva scritto nella swap la prima volta che questo veniva lanciato, e vi permaneva fino al riavvio della macchina (da questo il nome di *sticky bit*); essendo la swap un file conti-

⁷⁷anzi più precisamente un binario eseguibile: per motivi di sicurezza il kernel ignora i bit *suid* e *sgid* per gli script eseguibili.

nuo o una partizione indicizzata direttamente si poteva risparmiare in tempo di caricamento rispetto alla ricerca attraverso la struttura del filesystem. Lo *sticky bit* è indicato usando la lettera “t” al posto della “x” nei permessi per gli altri.

Ovviamente per evitare che gli utenti potessero intasare la swap solo l’amministratore era in grado di impostare questo bit, che venne chiamato anche con il nome di *saved text bit*, da cui deriva quello della costante. Le attuali implementazioni di memoria virtuale e filesystem rendono sostanzialmente inutile questo procedimento.

Benché ormai non venga più utilizzato per i file, lo *sticky bit* ha invece assunto un uso importante per le directory;⁷⁸ in questo caso se tale bit è impostato un file potrà essere rimosso dalla directory soltanto se l’utente ha il permesso di scrittura su di essa ed inoltre è vera una delle seguenti condizioni:

- l’utente è proprietario del file,
- l’utente è proprietario della directory,
- l’utente è l’amministratore.

Un classico esempio di directory che ha questo bit impostato è `/tmp`, i cui permessi infatti di solito sono i seguenti:

```
piccardi@hain:~/gapil$ ls -ld /tmp
drwxrwxrwt  6 root    root      1024 Aug 10 01:03 /tmp
```

quindi con lo *sticky bit* bit impostato. In questo modo qualunque utente nel sistema può creare dei file in questa directory, che come suggerisce il nome è normalmente utilizzata per la creazione di file temporanei, ma solo l’utente che ha creato un certo file potrà cancellarlo o rinominarlo. In questo modo si evita che un utente possa, più o meno consapevolmente, cancellare i file temporanei creati dagli altri utenti.

4.4.3 Le funzioni per la gestione dei permessi dei file

Come visto in sez. 4.4.1 il controllo di accesso ad un file viene fatto utilizzando l’*UID* ed il *GID* effettivo del processo; ci sono casi però in cui si può voler effettuare il controllo con l’*UID* reale ed il *GID* reale, vale a dire usando i valori di *UID* e *GID* relativi all’utente che ha lanciato il programma, e che, come accennato in sez. 4.4.2 e spiegato in dettaglio in sez. 3.2, non è detto siano uguali a quelli effettivi.

Per far questo si può usare la funzione di sistema `access`, il cui prototipo è:

```
#include <unistd.h>
int access(const char *pathname, int mode)
Verifica i permessi di accesso.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES l’accesso al file non è consentito, o non si ha il permesso di attraversare una delle directory di `pathname`.

EINVAL il valore di `mode` non è valido.

EROFS si è richiesto l’accesso in scrittura per un file su un filesystem montato in sola lettura.

ETXTBSY si è richiesto l’accesso in scrittura per un eseguibile binario correntemente in esecuzione.

ed inoltre EFAULT, EIO, ELOOP, ENAMETOOLONG, ENOENT, ENOTDIR nel loro significato generico.

⁷⁸lo *sticky bit* per le directory è un’estensione non definita nello standard POSIX, Linux però la supporta, così come BSD e SVr4.

La funzione verifica i permessi di accesso, indicati da `mode`, per il file indicato da `pathname`. I valori possibili per l'argomento `mode` sono esprimibili come combinazione delle costanti numeriche riportate in tab. 4.11 (attraverso un OR binario delle stesse). I primi tre valori implicano anche la verifica dell'esistenza del file, se si vuole verificare solo quest'ultima si può usare `F_OK`, o anche direttamente `stat`. Nel caso in cui `pathname` si riferisca ad un collegamento simbolico, questo viene seguito ed il controllo è fatto sul file a cui esso fa riferimento.

La funzione controlla solo i bit dei permessi di accesso per `pathname`, ma occorre poter risolvere quest'ultimo, e se non c'è il permesso di esecuzione per una qualunque delle sue componenti la funzione fallirà indipendentemente dai permessi del file. Si tenga presente poi che il fatto che una directory abbia permesso di scrittura non significa che vi si possa scrivere come fosse un file, e che se un file ha il permesso di esecuzione non è detto che sia eseguibile. La funzione ha successo solo se tutti i permessi controllati sono disponibili.

mode	Significato
R_OK	Verifica il permesso di lettura.
W_OK	Verifica il permesso di scrittura.
X_OK	Verifica il permesso di esecuzione.
F_OK	Verifica l'esistenza del file.

Tabella 4.11: Valori possibili per l'argomento `mode` della funzione `access`.

Un esempio tipico per l'uso di questa funzione è quello di un processo che sta eseguendo un programma coi privilegi di un altro utente (ad esempio attraverso l'uso del *suid bit*) che vuole controllare se l'utente originale ha i permessi per accedere ad un certo file, ma eseguire questo controllo prima di aprire il file espone al rischio di una *race condition* che apre ad un possibile *symlink attack* fra il controllo e l'apertura del file. In questo caso è sempre opportuno usare invece la funzione `faccessat` che tratteremo insieme alle altre *at-functions* in sez. 5.2.4.

Del tutto analoghe a `access` sono le due funzioni `euidaccess` e `eaccess` che ripetono lo stesso controllo usando però gli identificatori del gruppo effettivo, verificando quindi le effettive capacità di accesso ad un file. Le funzioni hanno entrambe lo stesso prototipo⁷⁹ che è del tutto identico a quello di `access`. Prendono anche gli stessi valori e restituiscono gli stessi risultati e gli stessi codici di errore.

Le due funzioni non sono previste da nessuno standard, ed utilizzabili solo avendo definito `_GNU_SOURCE`; inoltre qualora le si vogliano utilizzare per verificare che un processo abbia i permessi per accedere ad un file prima di farlo effettivamente, ci si esporrebbe ad una *race condition*, dato che i permessi potrebbero cambiare nell'intervallo fra il controllo e l'accesso effettivo. Per questo motivo in questo caso è molto più semplice e sicuro tentare direttamente l'accesso, e trattare opportunamente l'eventuale fallimento per mancanza di permessi.

Per cambiare i permessi di un file sono invece disponibili due funzioni di sistema `chmod` e `fchmod`, che operano rispettivamente usando il *pathname* di un file o se questo è già aperto sul relativo file descriptor; i loro prototipi sono:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *path, mode_t mode)
    Cambia i permessi del file indicato da path al valore indicato da mode.
int fchmod(int fd, mode_t mode)
    Analoga alla precedente, ma usa il file descriptor fd per indicare il file.
```

⁷⁹in realtà `eaccess` è solo un sinonimo di `euidaccess` fornita per compatibilità con l'uso di questo nome in altri sistemi.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EPERM l'`UID` effettivo non corrisponde a quello del proprietario del file o non si hanno i privilegi di amministratore.

EROFS il file è su un filesystem in sola lettura.

ed inoltre per entrambe `EIO`, per `chmod` `EACCES`, `EFAULT`, `ELOOP`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, per `fchmod` `EBADF` nel loro significato generico.

Entrambe le funzioni utilizzano come secondo argomento `mode`, una variabile dell'apposito tipo primitivo `mode_t` (vedi tab. 1.2), che è una maschera binaria da utilizzare per specificare i permessi sui file.

mode	Valore	Significato
<code>S_ISUID</code>	04000	Set user ID bit.
<code>S_ISGID</code>	02000	Set group ID bit.
<code>S_ISVTX</code>	01000	Sticky bit.
<code>S_IRWXU</code>	00700	L'utente ha tutti i permessi.
<code>S_IRUSR</code>	00400	L'utente ha il permesso di lettura.
<code>S_IWUSR</code>	00200	L'utente ha il permesso di scrittura.
<code>S_IXUSR</code>	00100	L'utente ha il permesso di esecuzione.
<code>S_IRWXG</code>	00070	Il gruppo ha tutti i permessi.
<code>S_IRGRP</code>	00040	Il gruppo ha il permesso di lettura.
<code>S_IWGRP</code>	00020	Il gruppo ha il permesso di scrittura.
<code>S_IXGRP</code>	00010	Il gruppo ha il permesso di esecuzione.
<code>S_IRWXO</code>	00007	Gli altri hanno tutti i permessi.
<code>S_IROTH</code>	00004	Gli altri hanno il permesso di lettura.
<code>S_IWOTH</code>	00002	Gli altri hanno il permesso di scrittura.
<code>S_IXOTH</code>	00001	Gli altri hanno il permesso di esecuzione.

Tabella 4.12: Valori delle costanti usate per indicare i vari bit di `mode` utilizzato per impostare i permessi dei file.

Si sono riportate in tab. 4.12 le costanti con cui indicare i singoli bit di `mode`; si noti come corrispondano agli stessi valori usati per `st_mode` già visti in tab. 4.7. Il valore di `mode` può essere ottenuto combinando fra loro con un OR binario le costanti simboliche relative ai vari bit, o specificato direttamente, come per l'omonimo comando di shell `chmod`, con un valore numerico (con la shell in genere lo si scrive in ottale, dato che i bit dei permessi sono divisibili in gruppi di tre), che si può calcolare direttamente usando lo schema di utilizzo dei bit illustrato in fig. 4.17.

Ad esempio i permessi standard assegnati ai nuovi file (lettura e scrittura per il proprietario, sola lettura per il gruppo e gli altri) sono corrispondenti al valore ottale `0644`, un programma invece avrebbe anche il bit di esecuzione attivo, con un valore ottale di `0755`, se infine si volesse attivare anche il bit `suid` il valore ottale da fornire sarebbe `4755`.

Il cambiamento dei permessi di un file eseguito attraverso queste funzioni ha comunque alcune limitazioni, previste per motivi di sicurezza. L'uso delle funzioni infatti è possibile solo se l'`UID` effettivo del processo chiamante corrisponde a quello del proprietario del file o se il processo ha i privilegi di amministratore,⁸⁰ in caso contrario esse falliranno con un errore di `EPERM`.

Ma oltre a questa regola generale, di immediata comprensione, esistono delle limitazioni ulteriori. Per questo motivo, anche se si è proprietari del file, non tutti i valori possibili di `mode` sono permessi o hanno effetto; in particolare accade che:

⁸⁰per la precisione la capacità `CAP_FOWNER`, vedi sez. 9.1.1.

1. siccome solo l'amministratore può impostare lo *sticky bit*, se l'*UID* effettivo del processo non è zero esso viene automaticamente cancellato, senza notifica di errore, qualora sia stato indicato in *mode*.
2. per quanto detto in sez. 4.4.4 riguardo la creazione dei nuovi file, si può avere il caso in cui il file creato da un processo è assegnato ad un gruppo per il quale il processo non ha privilegi. Per evitare che si possa assegnare il bit *sgid* ad un file appartenente ad un gruppo per cui non si hanno diritti, questo viene automaticamente cancellato da *mode*, senza notifica di errore, qualora il gruppo del file non corrisponda a quelli associati al processo; la cosa non avviene quando l'*UID* effettivo del processo è zero.

Per alcuni filesystem⁸¹ è inoltre prevista un'ulteriore misura di sicurezza, volta a scongiurare l'abuso dei bit *suid* e *sgid*; essa consiste nel cancellare automaticamente questi bit dai permessi di un file qualora un processo che non appartenga all'amministratore⁸² effettui una scrittura. In questo modo anche se un utente malizioso scopre un file *suid* su cui può scrivere, un'eventuale modifica comporterà la perdita di questo privilegio.

Le funzioni *chmod* e *fchmod* ci permettono di modificare i permessi di un file, resta però il problema di quali sono i permessi assegnati quando il file viene creato. Le funzioni dell'interfaccia nativa di Unix, come vedremo in sez. 5.1.2, permettono di indicare esplicitamente i permessi di creazione di un file, ma questo non è possibile per le funzioni dell'interfaccia standard ANSI C che non prevede l'esistenza di utenti e gruppi, ed inoltre il problema si pone anche per l'interfaccia nativa quando i permessi non vengono indicati esplicitamente.

Per le funzioni dell'interfaccia standard ANSI C l'unico riferimento possibile è quello della modalità di apertura del nuovo file (lettura/scrittura o sola lettura), che però può fornire un valore che è lo stesso per tutti e tre i permessi di sez. 4.4.1 (cioè 666 nel primo caso e 222 nel secondo). Per questo motivo il sistema associa ad ogni processo⁸³ una maschera di bit, la cosiddetta *umask*, che viene utilizzata per impedire che alcuni permessi possano essere assegnati ai nuovi file in sede di creazione. I bit indicati nella maschera vengono infatti cancellati dai permessi quando un nuovo file viene creato.⁸⁴

La funzione di sistema che permette di impostare il valore di questa maschera di controllo è *umask*, ed il suo prototipo è:

```
#include <stat.h>
mode_t umask(mode_t mask)
```

Imposta la maschera dei permessi.

La funzione ritorna il precedente valore della maschera, non sono previste condizioni di errore.

La funzione imposta la maschera dei permessi dei bit al valore specificato da *mask*, di cui vengono presi solo i 9 bit meno significativi. In genere si usa questa maschera per impostare un valore predefinito che escluda preventivamente alcuni permessi, il caso più comune è eliminare il permesso di scrittura per il gruppo e gli altri, corrispondente ad un valore per *mask* pari a 022. In questo modo è possibile cancellare automaticamente i permessi non voluti. Di norma questo valore viene impostato una volta per tutte al login (a 022 se non indicato altrimenti), e gli utenti non hanno motivi per modificarlo.

⁸¹i filesystem più comuni (*ext2*, *ext3*, *ext4*, *XFS*, *Btrfs*) supportano questa caratteristica, che è mutuata da BSD.

⁸²per la precisione un processo che non dispone della capacità *CAP_FSETID*, vedi sez. 9.1.1.

⁸³è infatti contenuta nel campo *umask* della struttura *fs_struct*, vedi fig. 3.2.

⁸⁴l'operazione viene fatta sempre: anche qualora si indichi esplicitamente un valore dei permessi nelle funzioni di creazione che lo consentono, i permessi contenuti nella *umask* verranno tolti.

4.4.4 La gestione della titolarità dei file

Vedremo in sez. 5.1.2 con quali funzioni si possono creare nuovi file, in tale occasione vedremo che è possibile specificare in sede di creazione quali permessi applicare ad un file, però non si può indicare a quale utente e gruppo esso deve appartenere. Lo stesso problema si presenta per la creazione di nuove directory (procedimento descritto in sez. 4.2.2).

Lo standard POSIX prescrive che l'*UID* del nuovo file corrisponda all'*UID* effettivo del processo che lo crea; per il *GID* invece prevede due diverse possibilità:

- che il *GID* del file corrisponda al *GID* effettivo del processo.
- che il *GID* del file corrisponda al *GID* della directory in cui esso è creato.

In genere BSD usa sempre la seconda possibilità, che viene per questo chiamata semantica BSD. Linux invece segue normalmente quella che viene chiamata semantica SVr4: di norma un nuovo file viene creato, seguendo la prima opzione, con il *GID* del processo, se però la directory in cui viene creato ha il bit *sgid* dei permessi impostato allora viene usata la seconda opzione. L'adozione di questa semantica però può essere controllata, all'interno di alcuni filesystem,⁸⁵ con l'uso dell'opzione di montaggio *grpuid*, che se attivata fa passare all'uso della semantica BSD.

Usare la semantica BSD ha il vantaggio che il *GID* viene sempre automaticamente propagato, restando coerente a quello della directory di partenza, in tutte le sotto-directory. La semantica SVr4 offre la possibilità di scegliere, ma per ottenere lo stesso risultato di coerenza che si ha con quella di BSD necessita che quando si creano nuove directory venga propagato il bit *sgid*. Questo è il comportamento predefinito del comando *mkdir*, ed è in questo modo ad esempio che le varie distribuzioni assicurano che le sotto-directory create nella home di un utente restino sempre con il *GID* del gruppo primario dello stesso.

La presenza del bit *sgid* è inoltre molto comoda quando si hanno directory contenenti file condivisi da un gruppo di utenti in cui possono scrivere tutti i membri dello stesso, dato che assicura che i file che gli utenti vi creano appartengano allo gruppo stesso. Questo non risolve però completamente i problemi di accesso da parte di altri utenti dello stesso gruppo, in quanto di default i permessi assegnati al gruppo non sono sufficienti per un accesso in scrittura; in questo caso si deve aver cura di usare prima della creazione dei file un valore per *umask* lasci il permesso di scrittura per il gruppo.⁸⁶

Come avviene nel caso dei permessi esistono anche delle appropriate funzioni di sistema, *chown*, *fchown* e *lchown*, che permettono di cambiare sia l'utente che il gruppo a cui un file appartiene; i rispettivi prototipi sono:

```
#include <sys/types.h>
#include <sys/stat.h>
int chown(const char *path, uid_t owner, gid_t group)
int fchown(int fd, uid_t owner, gid_t group)
int lchown(const char *path, uid_t owner, gid_t group)
```

Cambiano proprietario e gruppo proprietario di un file.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EPERM l'*UID* effettivo non corrisponde a quello del proprietario del file o non è zero, o utente e gruppo non sono validi.

ed inoltre per tutte *EROFS* e *EIO*, per *chown* *EACCES*, *EFAULT*, *ELOOP*, *ENAMETOOLONG*, *ENOENT*, *ENOMEM*, *ENOTDIR*, per *fchown* *EBADF* nel loro significato generico.

⁸⁵con il kernel 2.6.25 questi erano *ext2*, *ext3*, *ext4*, e *XFS*.

⁸⁶in tal caso si può assegnare agli utenti del gruppo una *umask* di 002, anche se la soluzione migliore in questo caso è usare una ACL di default (vedi sez. 4.5.2).

Le funzioni cambiano utente e gruppo di appartenenza di un file ai valori specificati dalle variabili `owner` e `group`. Con Linux solo l'amministratore⁸⁷ può cambiare il proprietario di un file; in questo viene seguita la semantica usata da BSD che non consente agli utenti di assegnare i loro file ad altri utenti evitando eventuali aggiramenti delle quote. L'amministratore può cambiare sempre il gruppo di un file, il proprietario può cambiare il gruppo solo dei file che gli appartengono e solo se il nuovo gruppo è il suo gruppo primario o uno dei gruppi di cui fa parte.

La funzione `chown` segue i collegamenti simbolici, per operare direttamente su un collegamento simbolico si deve usare la funzione `lchown`.⁸⁸ La funzione `fchown` opera su un file aperto, essa è mutuata da BSD, ma non è nello standard POSIX. Un'altra estensione rispetto allo standard POSIX è che specificando `-1` come valore per `owner` e `group` i valori restano immutati.

Quando queste funzioni sono chiamate con successo da un processo senza i privilegi di amministratore entrambi i bit `suid` e `sgid` vengono cancellati. Questo non avviene per il bit `sgid` nel caso in cui esso sia usato (in assenza del corrispondente permesso di esecuzione) per indicare che per il file è attivo il *mandatory locking* (vedi sez. 10.1.5).

4.4.5 Un quadro d'insieme sui permessi

Avendo affrontato in maniera separata il comportamento delle varie funzioni che operano sui permessi dei file ed avendo trattato in sezioni diverse il significato dei singoli bit dei permessi, vale la pena di fare un riepilogo in cui si riassumano le caratteristiche di ciascuno di essi, in modo da poter fornire un quadro d'insieme.

Nella parte superiore di tab. 4.13 si è riassunto il significato dei vari bit dei permessi per un file ordinario; per quanto riguarda l'applicazione dei permessi per proprietario, gruppo ed altri si ricordi quanto illustrato in sez. 4.4.1. Per compattezza, nella tabella si sono specificati i bit di *suid*, *sgid* e *sticky* con la notazione illustrata anche in fig. 4.17. Nella parte inferiore si sono invece riassunti i significati dei vari bit dei permessi per una directory; anche in questo caso si è riapplicato ai bit di *suid*, *sgid* e *sticky* la notazione illustrata in fig. 4.17.

Si ricordi infine che i permessi non hanno alcun significato per i collegamenti simbolici, mentre per i file di dispositivo hanno senso soltanto i permessi di lettura e scrittura, che si riflettono sulla possibilità di compiere dette operazioni sul dispositivo stesso.

Nella tabella si è indicato con il carattere “-” il fatto che il valore del bit in questione non è influente rispetto a quanto indicato nella riga della tabella; la descrizione del significato fa riferimento soltanto alla combinazione di bit per i quali è stato riportato esplicitamente un valore. Si rammenti infine che il valore dei bit dei permessi non ha alcun effetto qualora il processo possieda i privilegi di amministratore.

4.5 Caratteristiche e funzionalità avanzate

Tratteremo qui alcune caratteristiche e funzionalità avanzate della gestione di file e directory, affrontando anche una serie di estensioni dell'interfaccia classica dei sistemi unix-like, principalmente utilizzate a scopi di sicurezza, che sono state introdotte nelle versioni più recenti di Linux.

⁸⁷o in generale un processo con la capacità `CAP_CHOWN`, vedi sez. 9.1.1.

⁸⁸fino alla versione 2.1.81 in Linux `chown` non seguiva i collegamenti simbolici, da allora questo comportamento è stato assegnato alla funzione `lchown`, introdotta per l'occasione, ed è stata creata una nuova *system call* per `chown` che seguisse i collegamenti simbolici.

special			user			group			other			Significato per i file
s	s	t	r	w	x	r	w	x	r	w	x	
1	-	-	-	-	1	-	-	-	-	-	-	Eseguito conferisce l' <i>UID</i> effettivo dell'utente.
-	1	-	-	-	1	-	-	-	-	-	-	Eseguito conferisce il <i>GID</i> effettivo del gruppo.
-	1	-	-	-	0	-	-	-	-	-	-	Il <i>mandatory locking</i> è abilitato.
-	-	1	-	-	-	-	-	-	-	-	-	Non utilizzato.
-	-	-	1	-	-	-	-	-	-	-	-	Permesso di lettura per l'utente.
-	-	-	-	1	-	-	-	-	-	-	-	Permesso di scrittura per l'utente.
-	-	-	-	-	1	-	-	-	-	-	-	Permesso di esecuzione per l'utente.
-	-	-	-	-	-	1	-	-	-	-	-	Permesso di lettura per il gruppo.
-	-	-	-	-	-	-	1	-	-	-	-	Permesso di scrittura per il gruppo.
-	-	-	-	-	-	-	-	1	-	-	-	Permesso di esecuzione per il gruppo.
-	-	-	-	-	-	-	-	-	1	-	-	Permesso di lettura per tutti gli altri.
-	-	-	-	-	-	-	-	-	-	1	-	Permesso di scrittura per tutti gli altri.
-	-	-	-	-	-	-	-	-	-	-	1	Permesso di esecuzione per tutti gli altri.

special			user			group			other			Significato per le directory
s	s	t	r	w	x	r	w	x	r	w	x	
1	-	-	-	-	-	-	-	-	-	-	-	Non utilizzato.
-	1	-	-	-	-	-	-	-	-	-	-	Propaga il gruppo ai nuovi file creati.
-	-	1	-	-	-	-	-	-	-	-	-	Solo il proprietario di un file può rimuoverlo.
-	-	-	1	-	-	-	-	-	-	-	-	Permesso di visualizzazione per l'utente.
-	-	-	-	1	-	-	-	-	-	-	-	Permesso di aggiornamento per l'utente.
-	-	-	-	-	1	-	-	-	-	-	-	Permesso di attraversamento per l'utente.
-	-	-	-	-	-	1	-	-	-	-	-	Permesso di visualizzazione per il gruppo.
-	-	-	-	-	-	-	1	-	-	-	-	Permesso di aggiornamento per il gruppo.
-	-	-	-	-	-	-	-	1	-	-	-	Permesso di attraversamento per il gruppo.
-	-	-	-	-	-	-	-	-	1	-	-	Permesso di visualizzazione per tutti gli altri.
-	-	-	-	-	-	-	-	-	-	1	-	Permesso di aggiornamento per tutti gli altri.
-	-	-	-	-	-	-	-	-	-	-	1	Permesso di attraversamento per tutti gli altri.

Tabella 4.13: Tabella riassuntiva del significato dei bit dei permessi per un file e directory.

4.5.1 Gli attributi estesi

Nelle sezioni precedenti abbiamo trattato in dettaglio le varie informazioni che il sistema mantiene negli *inode*, e le varie funzioni che permettono di modificarle. Si sarà notato come in realtà queste informazioni siano estremamente ridotte. Questo è dovuto al fatto che Unix origina negli anni '70, quando le risorse di calcolo e di spazio disco erano minime. Con il venir meno di queste restrizioni è incominciata ad emergere l'esigenza di poter associare ai file delle ulteriori informazioni astratte (quelli che abbiamo chiamato genericamente *metadati*) che però non potevano trovare spazio nei dati classici mantenuti negli *inode*.

Per risolvere questo problema alcuni sistemi unix-like (e fra questi anche Linux) hanno introdotto un meccanismo generico, detto *Extended Attributes* che consente di associare delle informazioni ulteriori ai singoli file; ad esempio vengono usati per la gestione delle ACL, che tratteremo in sez. 4.5.2 e per le *file capabilities*, che vedremo in sez. 9.1.1. Gli *attributi estesi* (abbreviati in *xattr*) non sono altro che delle coppie nome/valore che sono associate permanentemente ad un oggetto sul filesystem, analoghi di quello che sono le variabili di ambiente (vedi sez. 2.3.3) per un processo.

Altri sistemi (come Solaris, MacOS e Windows) hanno adottato un meccanismo diverso in cui ad un file sono associati diversi flussi di dati, su cui possono essere mantenute ulteriori informazioni, che possono essere accedute con le normali operazioni di lettura e scrittura. Questi non vanno confusi con gli *Extended Attributes* (anche se su Solaris hanno lo stesso nome), che sono un meccanismo molto più semplice, che pur essendo limitato (potendo contenere solo una quantità limitata di informazione) hanno il grande vantaggio di essere molto più semplici

da realizzare, più efficienti,⁸⁹ e di garantire l'atomicità di tutte le operazioni.

In Linux gli attributi estesi sono sempre associati al singolo *inode* e l'accesso viene sempre eseguito in forma atomica, in lettura il valore corrente viene scritto su un buffer in memoria, mentre la scrittura prevede che ogni valore precedente sia sovrascritto.

Si tenga presente che non tutti i filesystem supportano gli *Extended Attributes*; al momento della scrittura di queste dispense essi sono presenti solo sui vari *extN*, *ReiserFS*, *JFS*, *XFS*, *Btrfs*, *Lustre* e *OCFS2*. Inoltre a seconda della implementazione ci possono essere dei limiti sulla quantità di attributi che si possono utilizzare.⁹⁰ Infine lo spazio utilizzato per mantenere gli attributi estesi viene tenuto in conto per il calcolo delle quote di utente e gruppo proprietari del file.

Come meccanismo per mantenere informazioni aggiuntive associate al singolo file, gli *Extended Attributes* possono avere usi anche molto diversi fra loro. Per poterli distinguere allora sono stati suddivisi in *classi*, a cui poter applicare requisiti diversi per l'accesso e la gestione. Per questo motivo il nome di un attributo deve essere sempre specificato nella forma `namespace.attribute`, dove `namespace` fa riferimento alla classe a cui l'attributo appartiene, mentre `attribute` è il nome ad esso assegnato. In tale forma il nome di un attributo esteso deve essere univoco. Al momento sono state definite le quattro classi di attributi riportate in tab. 4.14.

Nome	Descrizione
security	Gli <i>extended security attributes</i> : vengono utilizzati dalle estensioni di sicurezza del kernel (i <i>Linux Security Modules</i>), per le realizzazione di meccanismi evoluti di controllo di accesso come <i>SELinux</i> o le <i>capabilities</i> dei file di sez. 9.1.1.
system	Gli <i>extended system attributes</i> : sono usati dal kernel per memorizzare dati di sistema associati ai file come le ACL (vedi sez. 4.5.2) o le <i>capabilities</i> (vedi sez. 9.1.1).
trusted	I <i>trusted extended attributes</i> : vengono utilizzati per poter realizzare in <i>user space</i> meccanismi che consentano di mantenere delle informazioni sui file che non devono essere accessibili ai processi ordinari.
user	Gli <i>extended user attributes</i> : utilizzati per mantenere informazioni aggiuntive sui file (come il <i>mime-type</i> , la codifica dei caratteri o del file) accessibili dagli utenti.

Tabella 4.14: I nomi utilizzati valore di namespace per distinguere le varie classi di *Extended Attributes*.

Dato che uno degli usi degli *Extended Attributes* è di impiegarli per realizzare delle estensioni al tradizionale meccanismo dei controlli di accesso di Unix (come le ACL, *SELinux*, ecc.), l'accesso ai loro valori viene regolato in maniera diversa a seconda sia della loro classe che di quali, fra le estensioni che li utilizzano, sono poste in uso. In particolare, per ciascuna delle classi riportate in tab. 4.14, si hanno i seguenti casi:

security L'accesso agli *extended security attributes* dipende dalle politiche di sicurezza stabilite da loro stessi tramite l'utilizzo di un sistema di controllo basato sui *Linux Security Modules* (ad esempio *SELinux*). Pertanto l'accesso in lettura o scrittura dipende dalle politiche di sicurezza implementate all'interno dal modulo

⁸⁹cosa molto importante, specie per le applicazioni che richiedono una gran numero di accessi, come le ACL.

⁹⁰ad esempio nel caso di *ext2* ed *ext3* è richiesto che essi siano contenuti all'interno di un singolo blocco, pertanto con dimensioni massime pari a 1024, 2048 o 4096 byte a seconda delle dimensioni di quest'ultimo impostate in fase di creazione del filesystem, mentre con *XFS* e *Btrfs* non ci sono limiti ed i dati vengono memorizzati in maniera diversa (nell'*inode* stesso, in un blocco a parte, o in una struttura ad albero dedicata) per mantenerne la scalabilità; lasciamo i dettagli dei vari filesystem alla documentazione accessibile con `man xattr`.

di sicurezza che si sta utilizzando al momento (ciascuno avrà le sue). Se non è stato caricato nessun modulo di sicurezza l'accesso in lettura sarà consentito a tutti i processi, mentre quello in scrittura solo ai processi con privilegi amministrativi (per la precisione dotati della capacità `CAP_SYS_ADMIN`, vedi sez. 9.1.1).

system Anche l'accesso agli *extended system attributes* dipende dalle politiche di accesso che il kernel realizza anche utilizzando gli stessi valori in essi contenuti. Ad esempio nel caso delle ACL (vedi sez. 4.5.2) l'accesso è consentito in lettura ai processi che hanno la capacità di eseguire una ricerca sul file (cioè hanno il permesso di lettura sulla directory che contiene il file) ed in scrittura al proprietario del file o ai processi dotati della capacità `CAP_FOWNER`, vale a dire una politica di accesso analoga a quella impiegata per gli ordinari permessi dei file.

trusted L'accesso ai *trusted extended attributes*, sia per la lettura che per la scrittura, è consentito soltanto ai processi con privilegi amministrativi dotati della capacità `CAP_SYS_ADMIN`. In questo modo si possono utilizzare questi attributi per realizzare in *user space* dei meccanismi di controllo che accedono ad informazioni non disponibili ai processi ordinari.

user L'accesso agli *extended user attributes* è regolato dai normali permessi dei file: occorre avere il permesso di lettura per leggerli e quello di scrittura per scriverli o modificarli. Dato l'uso di questi attributi si è scelto di applicare al loro accesso gli stessi criteri che si usano per l'accesso al contenuto dei file (o delle directory) cui essi fanno riferimento. Questa scelta vale però soltanto per i file e le directory ordinarie, se valesse in generale infatti si avrebbe un serio problema di sicurezza dato che esistono diversi oggetti sul filesystem per i quali è normale avere il permesso di scrittura consentito a tutti gli utenti, come i collegamenti simbolici, o alcuni file di dispositivo come `/dev/null`. Se fosse possibile usare su di essi gli *extended user attributes* un utente qualunque potrebbe inserirvi dati a piacere.⁹¹

La semantica del controllo di accesso indicata inoltre non avrebbe alcun senso al di fuori di file e directory: i permessi di lettura e scrittura per un file di dispositivo attengono alle capacità di accesso al dispositivo sottostante,⁹² mentre per i collegamenti simbolici questi vengono semplicemente ignorati: in nessuno dei due casi hanno a che fare con il contenuto del file, e nella discussione relativa all'uso degli *extended user attributes* nessuno è mai stato capace di indicare una qualche forma sensata di utilizzo degli stessi per collegamenti simbolici o file di dispositivo, e neanche per le *fifo* o i *socket*.

Per questo motivo essi sono stati completamente disabilitati per tutto ciò che non sia un file regolare o una directory.⁹³ Inoltre per le directory è stata introdotta una ulteriore restrizione, dovuta di nuovo alla presenza ordinaria di permessi di scrittura completi su directory come `/tmp`. Per questo motivo, per evitare eventuali abusi, se una directory ha lo *sticky bit* attivo sarà consentito scrivere i suoi *extended user attributes* soltanto se si è proprietari della stessa, o si hanno i privilegi amministrativi della capacità `CAP_FOWNER`.

Le funzioni per la gestione degli attributi estesi, come altre funzioni di gestione avanzate specifiche di Linux, non fanno parte della *glibc*, e sono fornite da una apposita libreria,

⁹¹la cosa è stata notata su XFS, dove questo comportamento permetteva, non essendovi limiti sullo spazio occupabile dagli *Extended Attributes*, di bloccare il sistema riempiendo il disco.

⁹²motivo per cui si può formattare un disco anche se `/dev` è su un filesystem in sola lettura.

⁹³si può verificare la semantica adottata consultando il file `fs/xattr.c` dei sorgenti del kernel.

`libattr`, che deve essere installata a parte;⁹⁴ pertanto se un programma le utilizza si dovrà indicare esplicitamente l'uso della suddetta libreria invocando il compilatore con l'opzione `-lattr`.

Per leggere gli attributi estesi sono disponibili tre diverse funzioni di sistema, `getxattr`, `lgetxattr` e `fgetxattr`, che consentono rispettivamente di richiedere gli attributi relativi a un file, a un collegamento simbolico e ad un file descriptor; i rispettivi prototipi sono:

```
#include <sys/types.h>
#include <attr/xattr.h>
ssize_t getxattr(const char *path, const char *name, void *value, size_t size)
ssize_t lgetxattr(const char *path, const char *name, void *value, size_t size)
ssize_t fgetxattr(int filedes, const char *name, void *value, size_t size)
```

Leggono il valore di un attributo esteso.

Le funzioni ritornano un intero positivo che indica la dimensione dell'attributo richiesto in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`ENOATTR` l'attributo richiesto non esiste.

`ENOTSUP` gli attributi estesi non sono supportati dal filesystem o sono disabilitati.

`ERANGE` la dimensione `size` del buffer `value` non è sufficiente per contenere il risultato.

ed inoltre tutti gli errori delle analoghe della famiglia `stat` con lo stesso significato, ed in particolare `EPERM` se non si hanno i permessi di accesso all'attributo.

Le funzioni `getxattr` e `lgetxattr` prendono come primo argomento un *pathname* che indica il file di cui si vuole richiedere un attributo, la sola differenza è che la seconda, se il *pathname* indica un collegamento simbolico, restituisce gli attributi di quest'ultimo e non quelli del file a cui esso fa riferimento. La funzione `fgetxattr` prende invece come primo argomento un numero di file descriptor, e richiede gli attributi del file ad esso associato.

Tutte e tre le funzioni richiedono di specificare nell'argomento `name` il nome dell'attributo di cui si vuole ottenere il valore. Il nome deve essere indicato comprensivo di prefisso del *namespace* cui appartiene (uno dei valori di tab. 4.14) nella forma `namespace.attributenam`, come stringa terminata da un carattere NUL. Il valore dell'attributo richiesto verrà restituito nel buffer puntato dall'argomento `value` per una dimensione massima di `size` byte (gli attributi estesi possono essere costituiti arbitrariamente da dati testuali o binari); se quest'ultima non è sufficiente si avrà un errore di `ERANGE`.

Per evitare di dover indovinare la dimensione di un attributo per tentativi si può eseguire una interrogazione utilizzando un valore nullo per `size`; in questo caso non verrà letto nessun dato, ma verrà restituito come valore di ritorno della funzione chiamata la dimensione totale dell'attributo esteso richiesto, che si potrà usare come stima per allocare un buffer di dimensioni sufficienti.

Si tenga conto che questa è comunque una stima perché anche se le funzioni restituiscono la dimensione esatta dell'attributo al momento in cui sono eseguite, questa potrebbe essere modificata in qualunque momento da un successivo accesso eseguito da un altro processo, pertanto si verifichi sempre il valore di ritorno ed il codice di errore della funzione usata, senza dare per scontato che essa abbia sempre successo.

Un secondo gruppo di funzioni sono quelle che consentono di impostare il valore di un attributo esteso, le funzioni sono `setxattr`, `lsetxattr` e `fsetxattr` e consentono di operare rispettivamente su un file, su un collegamento simbolico o utilizzando un file descriptor; i rispettivi prototipi sono:

⁹⁴la versione corrente della libreria è `libattr1`.

```
#include <sys/types.h>
#include <attr/xattr.h>
int setxattr(const char *path, const char *name, const void *value,
            size_t size, int flags)
int lsetxattr(const char *path, const char *name, const void *value,
            size_t size, int flags)
int fsetxattr(int filedes, const char *name, const void *value,
            size_t size, int flags)
```

Impostano il valore di un attributo esteso.

Le funzioni ritornano un 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EEXIST` si è usato il flag `XATTR_CREATE` ma l'attributo esiste già.

`ENOATTR` si è usato il flag `XATTR_REPLACE` e l'attributo richiesto non esiste.

`ENOTSUP` gli attributi estesi non sono supportati dal filesystem o sono disabilitati.

ed inoltre tutti gli errori delle analoghe della famiglia `stat` con lo stesso significato ed in particolare `EPERM` se non si hanno i permessi di accesso all'attributo.

Le tre funzioni prendono come primo argomento un valore adeguato al loro scopo (un *pathname* le prime due, un file descriptor la terza), usato in maniera del tutto identica a quanto visto in precedenza per le analoghe che leggono gli attributi estesi.

Il secondo argomento, *name*, deve indicare, anche in questo caso con gli stessi criteri appena visti per le analoghe `getxattr`, `lgetxattr` e `fgetxattr`, il nome (completo di suffisso) dell'attributo su cui si vuole operare. Il valore che verrà assegnato all'attributo dovrà essere preparato nel buffer puntato da *value*, e la sua dimensione totale (in byte) dovrà essere indicata dall'argomento *size*.

Infine l'argomento *flag* consente di controllare le modalità di sovrascrittura dell'attributo esteso, esso può prendere due valori: con `XATTR_REPLACE` si richiede che l'attributo esista, nel qual caso verrà sovrascritto ed altrimenti si avrà errore; con `XATTR_CREATE` si richiede che l'attributo non esista, nel qual caso verrà creato, altrimenti si avrà errore ed il valore attuale non sarà modificato. Utilizzando per *flag* un valore nullo l'attributo verrà modificato se è già presente, o creato se non c'è.

Le funzioni finora illustrate permettono di leggere o scrivere gli attributi estesi presenti su un file, ma sarebbe altrettanto utile poter sapere quali sono questi attributi; per questo sono disponibili le ulteriori tre funzioni di sistema `listxattr`, `llistxattr` e `flistxattr` i cui prototipi sono:

```
#include <sys/types.h>
#include <attr/xattr.h>
ssize_t listxattr(const char *path, char *list, size_t size)
ssize_t llistxattr(const char *path, char *list, size_t size)
ssize_t flistxattr(int filedes, char *list, size_t size)
```

Leggono la lista degli attributi estesi di un file.

Le funzioni ritornano un intero positivo che indica la dimensione della lista in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`ENOTSUP` gli attributi estesi non sono supportati dal filesystem o sono disabilitati.

`ERANGE` la dimensione *size* del buffer *value* non è sufficiente per contenere il risultato.

ed inoltre tutti gli errori delle analoghe della famiglia `stat` con lo stesso significato ed in particolare `EPERM` se non si hanno i permessi di accesso all'attributo.

Come per le precedenti le tre funzioni leggono gli attributi rispettivamente di un file, un collegamento simbolico o specificando un file descriptor, da specificare con il loro primo argomento. Gli altri due argomenti, identici per tutte e tre, indicano rispettivamente il puntatore *list* al buffer dove deve essere letta la lista e la dimensione *size* di quest'ultimo.

La lista viene fornita come sequenza non ordinata dei nomi dei singoli attributi estesi (sempre comprensivi del prefisso della loro classe) ciascuno dei quali è terminato da un carattere nullo. I nomi sono inseriti nel buffer uno di seguito all'altro. Il valore di ritorno della funzione indica la dimensione totale della lista in byte.

Come per le funzioni di lettura dei singoli attributi se le dimensioni del buffer non sono sufficienti si avrà un errore, ma è possibile ottenere dal valore di ritorno della funzione una stima della dimensione totale della lista usando per `size` un valore nullo.

Infine per rimuovere semplicemente un attributo esteso, si ha a disposizione un ultimo gruppo di funzioni di sistema: `removexattr`, `lremovexattr` e `fremovexattr`; i rispettivi prototipi sono:

```
#include <sys/types.h>
#include <attr/xattr.h>
int removexattr(const char *path, const char *name)
int lremovexattr(const char *path, const char *name)
int fremovexattr(int filedes, const char *name)
```

Rimuovono un attributo esteso di un file.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`ENOATTR` l'attributo richiesto non esiste.

`ENOTSUP` gli attributi estesi non sono supportati dal filesystem o sono disabilitati.

ed inoltre tutti gli errori delle analoghe della famiglia `stat` con lo stesso significato ed in particolare `EPERM` se non si hanno i permessi di accesso all'attributo.

Le tre funzioni rimuovono un attributo esteso operando rispettivamente su di un file, su un collegamento simbolico o un file descriptor, che vengono specificati dal valore passato con il loro primo argomento. L'attributo da rimuovere deve essere anche in questo caso indicato con l'argomento `name` secondo le stesse modalità già illustrate in precedenza per le altre funzioni relative alla gestione degli attributi estesi.

4.5.2 Le Access Control List

Il modello classico dei permessi di Unix, per quanto funzionale ed efficiente, è comunque piuttosto limitato e per quanto possa aver coperto per lunghi anni le esigenze più comuni con un meccanismo semplice e potente, non è in grado di rispondere in maniera adeguata a situazioni che richiedono una gestione più complessa dei permessi di accesso.⁹⁵

Per questo motivo erano state progressivamente introdotte nelle varie versioni di Unix dei meccanismi di gestione dei permessi dei file più flessibili, nella forma delle cosiddette *Access Control List* (indicate usualmente con la sigla `ACL`). Nello sforzo di standardizzare queste funzionalità era stato creato un gruppo di lavoro il cui scopo era estendere lo standard `POSIX 1003` attraverso due nuovi insiemi di specifiche, la `POSIX 1003.1e` per l'interfaccia di programmazione e la `POSIX 1003.2c` per i comandi di shell.

Gli obiettivi del gruppo di lavoro erano però forse troppo ambizioni, e nel gennaio del 1998 i finanziamenti vennero ritirati senza che si fosse arrivati alla definizione dello standard richiesto. Dato però che una parte della documentazione prodotta era di alta qualità venne deciso di rilasciare al pubblico la diciassettesima bozza del documento, quella che va sotto il nome di *POSIX 1003.1e Draft 17*, che è divenuta la base sulla quale si definiscono le cosiddette *Posix ACL*.

⁹⁵già un requisito come quello di dare accesso in scrittura ad alcune persone ed in sola lettura ad altre non si può soddisfare in maniera semplice.

A differenza di altri sistemi, come ad esempio FreeBSD, nel caso di Linux si è scelto di realizzare le ACL attraverso l'uso degli *Extended Attributes* (appena trattati in sez. 4.5.1), e fornire tutte le relative funzioni di gestione tramite una libreria, `libacl` che nasconde i dettagli implementativi delle ACL e presenta ai programmi una interfaccia che fa riferimento allo standard POSIX 1003.1e.

Anche in questo caso le funzioni di questa libreria non fanno parte della *glibc* e devono essere installate a parte;⁹⁶ pertanto se un programma le utilizza si dovrà indicare esplicitamente l'uso della libreria `libacl` invocando il compilatore con l'opzione `-lacl`. Si tenga presente inoltre che le ACL devono essere attivate esplicitamente montando il filesystem⁹⁷ su cui le si vogliono utilizzare con l'opzione `acl` attiva. Dato che si tratta di una estensione è infatti opportuno utilizzarle soltanto laddove siano necessarie.

Una ACL è composta da un insieme di voci, e ciascuna voce è a sua volta costituita da un *tipo*, da un eventuale *qualificatore* (deve essere presente soltanto per le voci di tipo `ACL_USER` e `ACL_GROUP`) e da un insieme di permessi. Ad ogni oggetto sul filesystem si può associare una ACL che ne governa i permessi di accesso, detta *access ACL*. Inoltre per le directory si può impostare una ACL aggiuntiva, detta “*Default ACL*”, che serve ad indicare quale dovrà essere la ACL assegnata di default nella creazione di un file all'interno della directory stessa. Come avviene per i permessi le ACL possono essere impostate solo del proprietario del file, o da un processo con la capacità `CAP_FOWNER`.

Tipo	Descrizione
<code>ACL_USER_OBJ</code>	Voce che contiene i diritti di accesso del proprietario del file.
<code>ACL_USER</code>	Voce che contiene i diritti di accesso per l'utente indicato dal rispettivo qualificatore.
<code>ACL_GROUP_OBJ</code>	Voce che contiene i diritti di accesso del gruppo proprietario del file.
<code>ACL_GROUP</code>	Voce che contiene i diritti di accesso per il gruppo indicato dal rispettivo qualificatore.
<code>ACL_MASK</code>	Voce che contiene la maschera dei massimi permessi di accesso che possono essere garantiti da voci del tipo <code>ACL_USER</code> , <code>ACL_GROUP</code> e <code>ACL_GROUP_OBJ</code> .
<code>ACL_OTHER</code>	Voce che contiene i diritti di accesso di chi non corrisponde a nessuna altra voce dell'ACL.

Tabella 4.15: Le costanti che identificano i tipi delle voci di una ACL.

L'elenco dei vari tipi di voci presenti in una ACL, con una breve descrizione del relativo significato, è riportato in tab. 4.15. Tre di questi tipi, `ACL_USER_OBJ`, `ACL_GROUP_OBJ` e `ACL_OTHER`, corrispondono direttamente ai tre permessi ordinari dei file (proprietario, gruppo proprietario e tutti gli altri) e per questo una ACL valida deve sempre contenere una ed una sola voce per ciascuno di questi tipi.

Una ACL può poi contenere un numero arbitrario di voci di tipo `ACL_USER` e `ACL_GROUP`, ciascuna delle quali indicherà i permessi assegnati all'utente e al gruppo indicato dal relativo qualificatore. Ovviamente ciascuna di queste voci dovrà fare riferimento a un utente o a un gruppo diverso, e non corrispondenti a quelli proprietari del file. Inoltre se in una ACL esiste una voce di uno di questi due tipi, è obbligatoria anche la presenza di una ed una sola voce di tipo `ACL_MASK`, che negli altri casi è opzionale.

⁹⁶la versione corrente della libreria è `libacl1`, e nel caso si usi Debian la si può installare con il pacchetto omonimo e con il collegato `libacl1-dev` per i file di sviluppo.

⁹⁷che deve supportarle, ma questo è ormai vero per praticamente tutti i filesystem più comuni, con l'eccezione di NFS per il quale esiste però il supporto per le versioni NFSv2 e NFSv3 del protocollo, con NFSv4 esistono invece delle ACL native che hanno una semantica diversa, su di esse possono mappare le ACL POSIX, ma l'inverso è possibile solo in forma incompleta.

Una voce di tipo `ACL_MASK` serve a mantenere la maschera dei permessi che possono essere assegnati tramite voci di tipo `ACL_USER`, `ACL_GROUP` e `ACL_GROUP_OBJ`. Se in una di queste voci si fosse specificato un permesso non presente in `ACL_MASK` questo verrebbe ignorato. L'uso di una ACL di tipo `ACL_MASK` è di particolare utilità quando essa associata ad una *Default ACL* su una directory, in quanto i permessi così specificati verranno ereditati da tutti i file creati nella stessa directory. Si ottiene così una sorta di *umask* associata ad un oggetto sul filesystem piuttosto che a un processo.

Dato che le ACL vengono a costituire una estensione dei permessi ordinari, uno dei problemi che si erano posti nella loro standardizzazione era appunto quello della corrispondenza fra questi e le ACL. Come accennato i permessi ordinari vengono mappati nelle tre voci di tipo `ACL_USER_OBJ`, `ACL_GROUP_OBJ` e `ACL_OTHER` che devono essere presenti in qualunque ACL; un cambiamento ad una di queste voci viene automaticamente riflesso sui permessi ordinari dei file e viceversa.⁹⁸

In realtà la mappatura è diretta solo per le voci `ACL_USER_OBJ` e `ACL_OTHER`, nel caso di `ACL_GROUP_OBJ` questo vale soltanto se non è presente una voce di tipo `ACL_MASK`, che è quanto avviene normalmente se non sono presenti ACL aggiuntive rispetto ai permessi ordinari. Se invece questa è presente verranno tolti dai permessi di `ACL_GROUP_OBJ` (cioè dai permessi per il gruppo proprietario del file) tutti quelli non presenti in `ACL_MASK`.⁹⁹

Un secondo aspetto dell'incidenza delle ACL sul comportamento del sistema è quello relativo alla creazione di nuovi file,¹⁰⁰ che come accennato può essere modificato dalla presenza di una *Default ACL* sulla directory che andrà a contenerli. Se questa non c'è valgono le regole usuali illustrate in sez. 4.4.3, per cui essi sono determinati dalla *umask* del processo, e la sola differenza è che i permessi ordinari da esse risultanti vengono automaticamente rimappati anche su una ACL di accesso assegnata automaticamente al nuovo file, che contiene soltanto le tre corrispondenti voci di tipo `ACL_USER_OBJ`, `ACL_GROUP_OBJ` e `ACL_OTHER`.

Se invece è presente una ACL di default sulla directory che contiene il nuovo file, essa diventerà automaticamente anche la ACL di accesso di quest'ultimo, a meno di non aver indicato, nelle funzioni di creazione che lo consentono, uno specifico valore per i permessi ordinari.¹⁰¹ In tal caso saranno eliminati dalle voci corrispondenti che deriverebbero dalla ACL di default, tutti i permessi non presenti in tale indicazione.

Dato che questa è la ragione che ha portato alla loro creazione, la principale modifica introdotta nel sistema con la presenza della ACL è quella alle regole del controllo di accesso ai file che si sono illustrate in sez. 4.4.1. Come nel caso ordinario per il controllo vengono sempre utilizzati gli identificatori del gruppo *effective* del processo, ma in caso di presenza di una ACL sul file, i passi attraverso i quali viene stabilito se il processo ha il diritto di accesso sono i seguenti:

1. Se l'*UID* del processo è nullo (se cioè si è l'amministratore) l'accesso è sempre garantito senza nessun controllo.¹⁰²
2. Se l'*UID* del processo corrisponde al proprietario del file allora:

⁹⁸per permessi ordinari si intende quelli mantenuti nell'*inode*, che devono restare dato che un filesystem può essere montato senza abilitare le ACL.

⁹⁹questo diverso comportamento a seconda delle condizioni è stato introdotto dalla standardizzazione *POSIX 1003.1e Draft 17* per mantenere il comportamento invariato sui sistemi dotati di ACL per tutte quelle applicazioni che sono conformi soltanto all'ordinario standard *POSIX 1003.1*.

¹⁰⁰o oggetti sul filesystem, il comportamento discusso vale per le funzioni `open` e `creat` (vedi sez. 5.1.2), `mknod` (vedi sez. 4.2.2), `mknod` e `mkfifo` (vedi sez. 4.2.5).

¹⁰¹tutte le funzioni citate in precedenza supportano un argomento `mode` che indichi un insieme di permessi iniziale.

¹⁰²più precisamente se si devono avere le capacità `CAP_DAC_OVERRIDE` per i file e `CAP_DAC_READ_SEARCH` per le directory, vedi sez. 9.1.1.

- se la voce `ACL_USER_OBJ` contiene il permesso richiesto, l'accesso è consentito;
 - altrimenti l'accesso è negato.
3. Se l'*UID* del processo corrisponde ad un qualunque qualificatore presente in una voce `ACL_USER` allora:
 - se la voce `ACL_USER` corrispondente e la voce `ACL_MASK` contengono entrambe il permesso richiesto, l'accesso è consentito;
 - altrimenti l'accesso è negato.
 4. Se è il *GID* del processo o uno dei *GID* supplementari corrisponde al gruppo proprietario del file allora:
 - se la voce `ACL_GROUP_OBJ` e una eventuale voce `ACL_MASK` (se non vi sono voci di tipo `ACL_GROUP` questa può non essere presente) contengono entrambe il permesso richiesto, l'accesso è consentito;
 - altrimenti l'accesso è negato.
 5. Se è il *GID* del processo o uno dei *GID* supplementari corrisponde ad un qualunque qualificatore presente in una voce `ACL_GROUP` allora:
 - se la voce `ACL_GROUP` corrispondente e la voce `ACL_MASK` contengono entrambe il permesso richiesto, l'accesso è consentito;
 - altrimenti l'accesso è negato.
 6. Se la voce `ACL_USER_OBJ` contiene il permesso richiesto, l'accesso è consentito, altrimenti l'accesso è negato.

I passi di controllo vengono eseguiti esattamente in questa sequenza, e la decisione viene presa non appena viene trovata una corrispondenza con gli identificatori del processo. Questo significa che i permessi presenti in una voce di tipo `ACL_USER` hanno la precedenza sui permessi ordinari associati al gruppo proprietario del file (vale a dire su `ACL_GROUP_OBJ`).

Per la gestione delle ACL lo standard *POSIX 1003.1e Draft 17* ha previsto delle apposite funzioni ed tutta una serie di tipi di dati dedicati, arrivando fino a definire un tipo di dato e delle costanti apposite per identificare i permessi standard di lettura, scrittura ed esecuzione. Tutte le operazioni devono essere effettuate attraverso tramite questi tipi di dati, che incapsulano tutte le informazioni contenute nelle ACL.

La prima di queste funzioni che prendiamo in esame (si ricordi che come per tutte le altre per poterla usare occorre invocare il compilatore con l'opzione `-l acl`) è `acl_init`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_init(int count)
```

Inizializza un'area di lavoro per una ACL.

La funzione ritorna un oggetto di tipo `acl_t` in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` il valore di `count` è negativo.

`ENOMEM` non c'è sufficiente memoria disponibile.

La funzione alloca ed inizializza un'area di memoria che verrà usata per mantenere i dati di una ACL contenente fino ad un massimo di `count` voci. La funzione ritorna un valore di tipo `acl_t` da usare in tutte le altre funzioni che operano sulla ACL. La funzione si limita alla allocazione iniziale e non inserisce nessun valore nella ACL che resta vuota.

Si tenga presente che pur essendo `acl_t` un tipo opaco che identifica "l'oggetto" ACL, il valore restituito dalla funzione non è altro che un puntatore all'area di memoria allocata

per i dati richiesti. Pertanto in caso di fallimento verrà restituito un puntatore nullo di tipo “(acl_t) NULL” e si dovrà, in questa come in tutte le funzioni seguenti che restituiscono un oggetto di tipo `acl_t`, confrontare il valore di ritorno della funzione con NULL (anche se, a voler essere estremamente pignoli, si dovrebbe usare “(acl_t) NULL”, ma è sufficiente fare un confronto direttamente con NULL essendo cura del compilatore fare le conversioni necessarie).

Una volta che si siano completate le operazioni sui dati di una ACL la memoria allocata per un oggetto `acl_t` dovrà essere liberata esplicitamente attraverso una chiamata alla funzione `acl_free`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
int acl_free(void *obj_p)
```

Disalloca la memoria riservata per una ACL.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` può assumere solo il valore:

EINVAL `obj_p` non è valido.

Si noti come la funzione usi come argomento un puntatore di tipo “void *”, essa infatti può essere usata non solo per liberare la memoria allocata per i dati di una ACL, ma anche per quella usata per creare le stringhe di descrizione testuale delle ACL o per ottenere i valori dei qualificatori della una voce di una ACL. L’uso del tipo generico “void *” consente di evitare di eseguire un *cast* al tipo di dato di cui si vuole effettuare la deallocazione.

Si tenga presente poi che oltre a `acl_init` ci sono molte altre funzioni che possono allocare memoria per i dati delle ACL, è pertanto opportuno tenere traccia di tutte le chiamate a queste funzioni perché alla fine delle operazioni tutta la memoria allocata dovrà essere liberata con `acl_free`.

Una volta che si abbiano a disposizione i dati di una ACL tramite il riferimento ad oggetto di tipo `acl_t` questi potranno essere copiati con la funzione `acl_dup`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_dup(acl_t acl)
```

Crea una copia di una ACL.

La funzione ritorna un oggetto di tipo `acl_t` in caso di successo in caso di successo e NULL per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL l’argomento `acl` non è un puntatore valido per una ACL.

ENOMEM non c’è sufficiente memoria disponibile per eseguire la copia.

La funzione crea una copia dei dati della ACL indicata tramite l’argomento `acl`, allocando autonomamente tutto spazio necessario alla copia e restituendo un secondo oggetto di tipo `acl_t` come riferimento a quest’ultima. Valgono per questo le stesse considerazioni fatte per il valore di ritorno di `acl_init`, ed in particolare il fatto che occorrerà prevedere una ulteriore chiamata esplicita a `acl_free` per liberare la memoria occupata dalla copia.

Se si deve creare una ACL manualmente l’uso di `acl_init` è scomodo, dato che la funzione restituisce una ACL vuota, una alternativa allora è usare `acl_from_mode` che consente di creare una ACL a partire da un valore di permessi ordinari, il prototipo della funzione è:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_from_mode(mode_t mode)
```

Crea una ACL inizializzata con i permessi ordinari.

La funzione ritorna un oggetto di tipo `acl_t` in caso di successo e NULL per un errore, nel qual caso `errno` può assumere solo il valore ENOMEM.

La funzione restituisce una ACL inizializzata con le tre voci obbligatorie `ACL_USER_OBJ`, `ACL_GROUP_OBJ` e `ACL_OTHER` già impostate secondo la corrispondenza ai valori dei permessi ordinari indicati dalla maschera passata nell'argomento `mode`. Questa funzione è una estensione usata dalle ACL di Linux e non è portabile, ma consente di semplificare l'inizializzazione in maniera molto comoda.

Altre due funzioni che consentono di creare una ACL già inizializzata sono `acl_get_fd` e `acl_get_file`, che consentono di leggere la ACL di un file; i rispettivi prototipi sono:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_get_file(const char *path_p, acl_type_t type)
acl_t acl_get_fd(int fd)
```

Leggono i dati delle ACL di un file.

Le funzioni ritornano un oggetto di tipo `acl_t` in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EACCESS` non c'è accesso per una componente di `path_p` o si è richiesta una ACL di default per un file (solo per `acl_get_file`).

`EINVAL` `type` non ha un valore valido (solo per `acl_get_file`).

`ENOTSUP` il filesystem cui fa riferimento il file non supporta le ACL.

ed inoltre `ENOMEM` per entrambe, `EBADF` per `acl_get_fd`, e `ENAMETOOLONG`, `ENOENT`, `ENOTDIR`, per `acl_get_file` nel loro significato generico.

Le due funzioni ritornano, con un oggetto di tipo `acl_t`, il valore della ACL correntemente associata ad un file, che può essere identificato tramite un file descriptor usando `acl_get_fd` o con un *pathname* usando `acl_get_file`.

Nel caso di quest'ultima funzione, che può richiedere anche la ACL relativa ad una directory, il secondo argomento `type` consente di specificare se si vuole ottenere la ACL di default o quella di accesso. Questo argomento deve essere di tipo `acl_type_t` e può assumere solo i due valori riportati in tab. 4.16.

Tipo	Descrizione
<code>ACL_TYPE_ACCESS</code>	Indica una ACL di accesso.
<code>ACL_TYPE_DEFAULT</code>	Indica una ACL di default.

Tabella 4.16: Le costanti che identificano il tipo di ACL.

Si tenga presente che nel caso di `acl_get_file` occorrerà che il processo chiamante abbia privilegi di accesso sufficienti a poter leggere gli attributi estesi dei file (come illustrati in sez. 4.5.1); inoltre una ACL di tipo `ACL_TYPE_DEFAULT` potrà essere richiesta soltanto per una directory, e verrà restituita solo se presente, altrimenti verrà restituita una ACL vuota.

Infine si potrà creare una ACL direttamente dalla sua rappresentazione testuale con la funzione `acl_from_text`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_from_text(const char *buf_p)
```

Crea una ACL a partire dalla sua rappresentazione testuale.

La funzione ritorna un oggetto di tipo `acl_t` in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` la rappresentazione testuale all'indirizzo `buf_p` non è valida.

`ENOMEM` non c'è memoria sufficiente per allocare i dati.

La funzione prende come argomento il puntatore ad un buffer dove si è inserita la rappresentazione testuale della ACL che si vuole creare, la memoria necessaria viene automati-

camente allocata ed in caso di successo viene restituito come valore di ritorno un oggetto di tipo `acl_t` con il contenuto della stessa, che, come per le precedenti funzioni, dovrà essere deallocato esplicitamente al termine del suo utilizzo.

La rappresentazione testuale di una ACL è quella usata anche dai comandi ordinari per la gestione delle ACL (`getfacl` e `setfacl`), che prevede due diverse forme, estesa e breve, entrambe supportate da `acl_from_text`. La forma estesa prevede che sia specificata una voce per riga, nella forma:

```
tipo:qualificatore:permessi
```

dove il tipo può essere uno fra `user`, `group`, `other` e `mask`. Il qualificatore è presente solo per `user` e `group` ed indica l'utente o il gruppo a cui la voce si riferisce, mentre i permessi sono espressi con una tripletta di lettere analoga a quella usata per i permessi dei file, vale a dire “r” per il permesso di lettura, “w” per il permesso di scrittura, “x” per il permesso di esecuzione (scritti in quest'ordine) e “-” per l'assenza del permesso.

Un possibile esempio di rappresentazione della ACL di un file ordinario a cui, oltre ai permessi ordinari, si è aggiunto un altro utente con un accesso in lettura, è il seguente:

```
user::rw-
group::r--
other::r--
user:piccardi:r--
group:gabil:r--
```

Va precisato che i due tipi `user` e `group` sono usati rispettivamente per indicare delle voci relative ad utenti e gruppi (cioè per voci di tipo `ACL_USER_OBJ` e `ACL_USER` per `user` e `ACL_GROUP_OBJ` e `ACL_GROUP` per `group`) applicate sia a quelli proprietari del file che a quelli generici. Quelle dei proprietari si riconoscono per l'assenza di un qualificatore, ed in genere si scrivono per prima delle altre. Il significato delle voci di tipo `mask` e `mark` è evidente. Usando questa forma estesa si possono anche inserire dei commenti nel testo precedendoli con il carattere “#”.

La forma breve prevede invece la scrittura delle singole voci su una riga, separate da virgole; come specificatori del tipo di voce si possono usare le iniziali dei valori usati nella forma estesa (cioè “u”, “g”, “o” e “m”), mentre le altre parti della voce sono le stesse. In questo caso non sono consentiti permessi.

Per la conversione inversa, che consente di ottenere la rappresentazione testuale di una ACL, sono invece disponibili due funzioni. La prima delle due, di uso più immediato, è `acl_to_text`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
char *acl_to_text(acl_t acl, ssize_t *len_p)
                                Produce la rappresentazione testuale di una ACL.
```

La funzione ritorna il puntatore ad una stringa con la rappresentazione testuale della ACL in caso di successo e NULL per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` o `acl` non è un puntatore ad una ACL o la ACL che esso indica non è valida o non può esser tradotta in forma testuale.

`ENOMEM` non c'è memoria sufficiente per allocare i dati.

La funzione restituisce il puntatore ad una stringa, terminata da un NUL, contenente la rappresentazione testuale in forma estesa della ACL passata come argomento, ed alloca automaticamente la memoria necessaria. Questa dovrà poi essere liberata, quando non più necessaria, con `acl_free`. Se nell'argomento `len_p` si passa come valore il puntatore ad una

variabile intera, in questa verrà restituita (come *value result argument*) la dimensione della stringa con la rappresentazione testuale, non comprendente il carattere nullo finale.

La seconda funzione, che permette di controllare con una gran dovizia di particolari la generazione della stringa contenente la rappresentazione testuale della ACL, è `acl_to_any_text`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
char *acl_to_any_text(acl_t acl, const char *prefix, char separator, int options)
                                Produce la rappresentazione testuale di una ACL.
```

La funzione ritorna il puntatore ad una stringa con la rappresentazione testuale della ACL in caso di successo e NULL per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` la ACL indicata da `acl` non è valida.

`ENOMEM` non c'è memoria sufficiente per allocare i dati.

La funzione converte in formato testo la ACL indicata dall'argomento `acl`, usando il carattere `separator` come separatore delle singole voci; se l'argomento `prefix` non è nullo la stringa da esso indicata viene utilizzata come prefisso per le singole voci.

L'ultimo argomento, `options`, consente di controllare la modalità con cui viene generata la rappresentazione testuale. Un valore nullo fa sì che vengano usati gli identificatori standard `user`, `group`, `other` e `mask` con i nomi di utenti e gruppi risolti rispetto ai loro valori numerici. Altrimenti si può specificare un valore in forma di maschera binaria, da ottenere con un OR aritmetico dei valori riportati in tab. 4.17.

Tipo	Descrizione
<code>TEXT_ABBREVIATE</code>	Stampa le voci in forma abbreviata.
<code>TEXT_NUMERIC_IDS</code>	Non effettua la risoluzione di <code>UID</code> e <code>GID</code> lasciando i valori numerici.
<code>TEXT_SOME_EFFECTIVE</code>	Per ciascuna voce che contiene permessi che vengono eliminati dalla <code>ACL_MASK</code> viene generato un commento con i permessi effettivamente risultanti; il commento è separato con un tabulatore.
<code>TEXT_ALL_EFFECTIVE</code>	Viene generato un commento con i permessi effettivi per ciascuna voce che contiene permessi citati nella <code>ACL_MASK</code> , anche quando questi non vengono modificati da essa; il commento è separato con un tabulatore.
<code>TEXT_SMART_INDENT</code>	Da usare in combinazione con le precedenti opzioni <code>TEXT_SOME_EFFECTIVE</code> e <code>TEXT_ALL_EFFECTIVE</code> , aumenta automaticamente il numero di spaziatori prima degli eventuali commenti in modo da mantenerli allineati.

Tabella 4.17: Possibili valori per l'argomento `options` di `acl_to_any_text`.

Come per `acl_to_text` anche in questo caso il buffer contenente la rappresentazione testuale dell'ACL, di cui la funzione restituisce l'indirizzo, viene allocato automaticamente, e dovrà essere esplicitamente disallocato con una chiamata ad `acl_free`. Si tenga presente infine che questa funzione è una estensione specifica di Linux, e non è presente nella bozza dello standard POSIX.1e.

Per quanto utile per la visualizzazione o l'impostazione da riga di comando delle ACL, la forma testuale non è la più efficiente per poter memorizzare i dati relativi ad una ACL, ad esempio quando si vuole eseguirne una copia a scopo di archiviazione. Per questo è stata prevista la possibilità di utilizzare una rappresentazione delle ACL in una apposita forma binaria contigua e persistente. È così possibile copiare il valore di una ACL in un buffer e da questa rappresentazione tornare indietro e generare una ACL.

Lo standard POSIX.1e prevede a tale scopo tre funzioni, la prima e più semplice è `acl_size`, che consente di ottenere la dimensione che avrà la citata rappresentazione binaria, in modo da poter allocare per essa un buffer di dimensione sufficiente, il suo prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
ssize_t acl_size(acl_t acl)
    Determina la dimensione della rappresentazione binaria di una ACL.
```

La funzione ritorna la dimensione in byte della rappresentazione binaria della ACL in caso di successo e `-1` per un errore, nel qual caso `errno` può assumere solo il valore:
EINVAL la ACL indicata da `acl` non è valida.

Ottenuta con `acl_size` la dimensione del buffer necessaria per potervi memorizzare una ACL questo dovrà potrà essere allocato direttamente con `malloc`, ed a questo punto vi si potrà salvare la rappresentazione binaria della precedente ACL utilizzando la funzione `acl_copy_ext`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
ssize_t acl_copy_ext(void *buf_p, acl_t acl, ssize_t size)
    Ottiene la rappresentazione binaria di una ACL.
```

La funzione ritorna la dimensione in byte della rappresentazione binaria della ACL in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:
EINVAL la ACL indicata da `acl` non è valida, o `acl` non è un puntatore ad una ACL o `size` è negativo o nullo.
ERANGE il valore di `size` è più piccolo della dimensione della rappresentazione binaria della ACL.

La funzione scriverà la rappresentazione binaria della ACL indicata da `acl` sul buffer di dimensione `size` all'indirizzo `buf_p`, restituendo la dimensione della stessa come valore di ritorno. Qualora la dimensione della rappresentazione ecceda il valore di `size` la funzione fallirà con un errore di **ERANGE**. La funzione non ha nessun effetto sulla ACL indicata da `acl`.

Viceversa se si vuole ripristinare una ACL a partire da una rappresentazione binaria si potrà usare la funzione `acl_copy_int`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/acl.h>
acl_t acl_copy_int(const void *buf_p)
    Ripristina la rappresentazione binaria di una ACL.
```

La funzione ritorna un oggetto di tipo `acl_t` in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:
EINVAL il buffer all'indirizzo `buf_p` non contiene una rappresentazione corretta di una ACL.
ENOMEM non c'è memoria sufficiente per allocare un oggetto `acl_t` per la ACL richiesta.

La funzione alloca autonomamente un oggetto di tipo `acl_t`, restituito come valore di ritorno, con il contenuto della ACL rappresentata dai dati del buffer puntato da `buf_p`. Al solito l'oggetto `acl_t` dovrà essere deallocato esplicitamente al termine del suo utilizzo.

Una volta che si disponga della ACL desiderata, questa potrà essere impostata su un file o una directory. Per impostare una ACL sono disponibili due funzioni: `acl_set_file`, che opera sia su file che su directory usando un *pathname*, e `acl_set_fd` che opera solo su file usando un file descriptor; i rispettivi prototipi sono:


```
#include <sys/types.h>
#include <sys/acl.h>
int acl_set_file(const char *path, acl_type_t type, acl_t acl)
                                                    Imposta una ACL su un file o una directory.
int acl_set_fd(int fd, acl_t acl)
                                                    Imposta una ACL su un file descriptor.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES o un generico errore di accesso a `path` o il valore di `type` specifica una ACL il cui tipo non può essere assegnato a `path`.

EINVAL o `acl` non è una ACL valida, o `type` ha un valore non corretto per `acl_set_file` o o ha più voci di quante se ne possono assegnare al file per `acl_set_fd`.

ENOSPC non c'è spazio disco sufficiente per contenere i dati aggiuntivi della ACL.

ENOTSUP si è cercato di impostare una ACL su un file contenuto in un filesystem che non supporta le ACL.

ed inoltre nel loro significato generico EPERM, EROFS per entrambe, EBADF per `acl_set_fd`, ENAMETOOLONG, ENOENT, ENOTDIR per `acl_set_file`.

Con `acl_set_file` si assegna la ACL contenuta in `acl` al file o alla directory indicate da `path`, con `type` che indica il tipo di ACL con le costanti di tab. 4.16; si tenga presente però che le ACL di default possono essere solo impostate qualora `path` indichi una directory. Inoltre perché la funzione abbia successo la ACL dovrà essere valida, e contenere tutti le voci necessarie, con l'eccezione si specifica una ACL vuota per cancellare la ACL di default associata a `path`, valida solo in caso di directory.¹⁰³

La seconda funzione, `acl_set_fd`, è del tutto è analoga alla prima, ma non dovendo avere a che fare con directory (e la conseguente possibilità di avere una ACL di default) non necessita che si specifichi il tipo di ACL, che sarà sempre di accesso, e prende come unico argomento, a parte il file descriptor, la ACL da impostare.

Le funzioni viste finora operano a livello di una intera ACL, eseguendo in una sola volta tutte le operazioni relative a tutte le voci in essa contenuta. In generale è possibile modificare un singolo valore all'interno di una singola voce direttamente con le funzioni previste dallo standard POSIX.1e. Queste funzioni però sono alquanto macchinose da utilizzare per cui è molto più semplice operare direttamente sulla rappresentazione testuale. Questo è il motivo per non tratteremo nei dettagli dette funzioni, fornendone solo una descrizione sommaria; chi fosse interessato può ricorrere alle pagine di manuale.

Se si vuole operare direttamente sui contenuti di un oggetto di tipo `acl_t` infatti occorre fare riferimento alle singole voci tramite gli opportuni puntatori di tipo `acl_entry_t`, che possono essere ottenuti dalla funzione `acl_get_entry` (per una voce esistente) o dalla funzione `acl_create_entry` per una voce da aggiungere. Nel caso della prima funzione si potrà poi ripetere la lettura per ottenere i puntatori alle singole voci successive alla prima.

Una volta ottenuti detti puntatori si potrà operare sui contenuti delle singole voci: con le funzioni `acl_get_tag_type`, `acl_get_qualifier`, `acl_get_permset` si potranno leggere rispettivamente tipo, qualificatore e permessi, mentre con le corrispondenti `acl_set_tag_type`, `acl_set_qualifier`, `acl_set_permset` si potranno impostare i valori; in entrambi i casi vengono utilizzati tipi di dato ad hoc, descritti nelle pagine di manuale. Si possono poi copiare i valori di una voce da una ACL ad un'altra con `acl_copy_entry` o eliminare una voce da una ACL con `acl_delete_entry` e verificarne la validità prima di usarla con `acl_valid` o `acl_check`.

¹⁰³ questo però è una estensione della implementazione delle ACL di Linux, la bozza di standard POSIX.1e prevedeva l'uso della apposita funzione `acl_delete_def_file`, che prende come unico argomento il *pathname* della directory di cui si vuole cancellare l'ACL di default, per i dettagli si ricorra alla pagina di manuale.

```

1 #include <sys/types.h>    /* primitive system data types */
2 #include <stdlib.h>       /* C standard library */
3 #include <stdio.h>        /* standard I/O library */
4 #include <unistd.h>       /* unix standard library */
5 #include <sys/acl.h>      /* acl library (use -l acl) */
6
7 int main(int argc, char *argv[])
8 {
9     /*
10      * Variables definition
11      */
12     acl_t acl;
13     ssize_t size;
14     char * buffer;
15     ...
16     /* must have an argument */
17     if ((argc - optind) != 1) {
18         printf("Wrong_number_of_arguments_%d\n", argc - optind);
19         usage();
20     }
21     /* main body */
22     acl = acl_get_file(argv[1], ACL_TYPE_ACCESS);
23     if (acl == NULL) {
24         perror("cannot_get_acl_for_file");
25         return 1;
26     }
27     buffer = acl_to_text(acl, &size);
28     if (buffer == NULL) {
29         perror("cannot_convert_acl");
30         return 1;
31     }
32     printf("ACL_for_file_%s':\n%s\n", argv[1], buffer);
33     acl_free(acl);
34     acl_free(buffer);
35     return 0;
36 }

```

Figura 4.18: Corpo principale del programma `mygetfacl.c`.

Come esempio di utilizzo di queste funzioni nei sorgenti allegati alla guida si è distribuito il programma `mygetfacl.c`, che consente di leggere le ACL di un file, passato come argomento.

La sezione principale del programma, da cui si è rimossa la sezione sulla gestione delle opzioni, è riportata in fig. 4.18. Il programma richiede un unico argomento (16-20) che indica il file di cui si vuole leggere la ACL. Se questo è presente si usa (22) la funzione `get_acl_file` per leggerne la ACL, e si controlla (23-26) se l'operazione ha successo, uscendo con un messaggio di errore in caso contrario.

Ottenuta la ACL la si converte in formato testuale (27) con la funzione `acl_to_text`, controllando di nuovo se l'operazione ha successo (28-31) ed uscendo in caso contrario. Si provvede infine a stampare la rappresentazione testuale (32) e dopo aver liberato (33-34) le risorse allocate automaticamente, si conclude l'esecuzione.

4.5.3 La gestione delle quote disco

Quella delle quote disco è una funzionalità introdotta inizialmente da BSD e presente in Linux fino dai kernel dalla serie 2.0, che consente di porre dei tetti massimi al consumo delle risorse di un filesystem (spazio disco e *inode*) da parte di utenti e gruppi.

Dato che la funzionalità ha senso solo per i filesystem su cui si mantengono i dati degli utenti essa deve essere attivata esplicitamente.¹⁰⁴ Questo si fa, per tutti i filesystem che le supportano, tramite due distinte opzioni di montaggio, *usrquota* e *grpquota*, che abilitano le quote rispettivamente per gli utenti e per i gruppi. Così è possibile usare le limitazioni delle quote o sugli utenti o sui gruppi o su entrambi.

Dal kernel 4.1, ed inizialmente solo per il filesystem XFS, sono diventate disponibili un terzo tipo di quote, dette *project quota*, che consentono di applicare delle quote ad un “*project*”, identificato come ramo di albero sotto una directory, per il quale possono essere imposti dei limiti (di nuovo in termini di spazio disco o *inode*) per i file che ne fanno parte. Si può così porre dei limiti sul contenuto di un ramo di albero.

Il meccanismo prevede che per ciascun filesystem che supporta le quote disco (i vari *extN*, *Btrfs*, *XFS*, *JFS*, *ReiserFS*) il kernel provveda sia a mantenere aggiornati i dati relativi al consumo delle risorse da parte degli utenti e dei gruppi (e del progetto), che a far rispettare i limiti imposti dal sistema, con la generazione di un errore di *EDQUOT* per tutte le operazioni sui file che porterebbero ad un superamento degli stessi. Si tenga presente che questi due compiti sono separati, il primo si attiva al montaggio del filesystem con il supporto per le quote, il secondo deve essere abilitato esplicitamente.

Per il mantenimento dei dati di consumo delle risorse vengono usati due file riservati nella directory radice del filesystem su cui si sono attivate le quote, uno per le quote utente e l'altro per le quote gruppo.¹⁰⁵ Con la versione 2 del supporto delle quote, che da anni è l'unica rimasta in uso, questi file sono *aquota.user* e *aquota.group*, in precedenza erano *quota.user* e *quota.group*.

Dato che questi file vengono aggiornati soltanto se il filesystem è stato montato attivando il supporto delle quote, se si abilita il supporto in un secondo tempo e nel frattempo sono state eseguite delle operazioni sul filesystem quando il supporto era disabilitato, i dati contenuti possono non corrispondere esattamente allo stato corrente del consumo delle risorse. Per questo motivo prima di montare in scrittura un filesystem su cui sono abilitate le quote viene richiesto di utilizzare il comando *quotacheck* per verificare e aggiornare i dati.

Le restrizioni sul consumo delle risorse previste dal sistema delle quote prevedono sempre la presenza di due diversi limiti, il primo viene detto *soft limit* e può essere superato per brevi periodi di tempo senza che causare errori per lo sfioramento delle quote, il secondo viene detto *hard limit* e non può mai essere superato.

Il periodo di tempo per cui è possibile eccedere rispetto alle restrizioni indicate dal *soft limit* è detto “*periodo di grazia*” (*grace period*), che si attiva non appena si supera la quota da esso indicata. Se si continua a restare al di sopra del *soft limit* una volta scaduto il *grace period* questo verrà trattato allo stesso modo dell'*hard limit* e si avrà l'emissione immediata di un errore.

Si tenga presente infine che entrambi i tipi di limiti (*soft limit* e *hard limit*) possono essere disposti separatamente su entrambe le risorse di un filesystem, essi cioè possono essere presenti

¹⁰⁴in genere la si attiva sul filesystem che contiene le *home* degli utenti, dato che non avrebbe senso per i file di sistema che in genere appartengono all'amministratore.

¹⁰⁵la cosa vale per tutti i filesystem tranne *XFS* che mantiene i dati internamente, compresi quelli per le *project quota*, che pertanto, essendo questo l'unico filesystem che le supporta, non hanno un file ad esse riservato.

in maniera indipendente sia sullo spazio disco, con un massimo per il numero di blocchi, che sui file, con un massimo per il numero di *inode*.

La funzione di sistema che consente di controllare tutti i vari aspetti della gestione delle quote è `quotactl`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/quota.h>
int quotactl(int cmd, const char *dev, int id, caddr_t addr)
    Esegue una operazione di controllo sulle quote disco.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES	si è richiesto Q_QUOTAON, ma il file delle quote indicato da <code>addr</code> esiste ma non è un file ordinario o non sta su <code>dev</code> .
EBUSY	si è richiesto Q_QUOTAON, ma le quote sono già attive.
EFAULT	<code>addr</code> o <code>dev</code> non sono un puntatori validi.
EINVAL	o <code>cmd</code> non è un comando valido, o il dispositivo <code>dev</code> non esiste.
EIO	errore di lettura/scrittura sul file delle quote.
EMFILE	non si può aprire il file delle quote avendo superato il limite sul numero di file aperti nel sistema.
ENODEV	<code>dev</code> non corrisponde ad un <i>mount point</i> attivo.
ENOPKG	il kernel è stato compilato senza supporto per le quote.
ENOTBLK	<code>dev</code> non è un dispositivo a blocchi.
EPERM	non si hanno i permessi per l'operazione richiesta.
ESRCH	è stato richiesto uno fra Q_GETQUOTA, Q_SETQUOTA, Q_SETUSE, Q_SETQLIM per un filesystem senza quote attivate.

La funzione richiede che il filesystem sul quale si vuole operare, che deve essere specificato con il nome del relativo file di dispositivo nell'argomento `dev`, sia montato con il supporto delle quote abilitato. Per le operazioni che lo richiedono inoltre si dovrà indicare con l'argomento `id` l'utente o il gruppo o il progetto (specificati rispettivamente per *UID*, *GID* o identificativo) su cui si vuole operare,¹⁰⁶ o altri dati relativi all'operazione. Alcune operazioni più complesse usano infine l'argomento `addr` per indicare un indirizzo ad un area di memoria il cui utilizzo dipende dall'operazione stessa.

La funzione prevede la possibilità di eseguire una serie operazioni sulle quote molto diverse fra loro, la scelta viene effettuata tramite il primo argomento, `cmd`, che però oltre all'operazione indica anche a quale tipo di quota (utente o gruppo) l'operazione deve applicarsi. Per questo il valore di questo argomento viene costruito con l'ausilio della di una apposita macro `QCMD`:

```
#include <sys/quota.h>
int QCMD(subcmd, type)
    Imposta il comando subcmd per il tipo di quote (utente o gruppo) type.
```

La macro consente di specificare, oltre al tipo di operazione, da indicare con l'argomento `subcmd`, se questa deve applicarsi alle quote utente o alle quote gruppo o alle quote progetto. Questo viene indicato dall'argomento `type` che deve essere sempre definito ed assegnato ad uno fra i valori `USRQUOTA`, `GRPQUOTA` e `PRJQUOTA`.

I possibili valori per l'argomento `subcmd` di `QCMD` sono riportati in tab. 4.18, che illustra brevemente il significato delle operazioni associate a ciascuno di essi. In generale le operazioni di attivazione, disattivazione e di modifica dei limiti delle quote sono riservate e richiedono i

¹⁰⁶nel caso di *project quota* gli identificativi vengono associati alla directory base del progetto nel file `/etc/projects`, ed impostati con `xfs_quota`, l'argomento è di natura sistemistica e va al di là dello scopo di questo testo.

Comando	Descrizione
Q_QUOTAON	Attiva l'applicazione delle quote disco per il filesystem indicato da dev, si deve passare in addr il <i>pathname</i> al file che mantiene le quote, che deve esistere, e id deve indicare la versione del formato con uno dei valori di tab. 4.20; l'operazione richiede i privilegi di amministratore.
Q_QUOTAOFF	Disattiva l'applicazione delle quote disco per il filesystem indicato da dev, id e addr vengono ignorati; l'operazione richiede i privilegi di amministratore.
Q_GETQUOTA	Legge i limiti ed i valori correnti delle quote nel filesystem indicato da dev per l'utente o il gruppo specificato da id; si devono avere i privilegi di amministratore per leggere i dati relativi ad altri utenti o a gruppi di cui non si fa parte, il risultato viene restituito in una struttura dqblk all'indirizzo indicato da addr.
Q_SETQUOTA	Imposta i limiti per le quote nel filesystem indicato da dev per l'utente o il gruppo specificato da id secondo i valori ottenuti dalla struttura dqblk puntata da addr; l'operazione richiede i privilegi di amministratore.
Q_GETINFO	Legge le informazioni (in sostanza i <i>grace time</i>) delle quote del filesystem indicato da dev sulla struttura dqinfo puntata da addr, id viene ignorato.
Q_SETINFO	Imposta le informazioni delle quote del filesystem indicato da dev come ottenuti dalla struttura dqinfo puntata da addr, id viene ignorato; l'operazione richiede i privilegi di amministratore.
Q_GETFMT	Richiede il valore identificativo (quello di tab. 4.20) per il formato delle quote attualmente in uso sul filesystem indicato da dev, che sarà memorizzato sul buffer di 4 byte puntato da addr.
Q_SYNC	Aggiorna la copia su disco dei dati delle quote del filesystem indicato da dev; in questo caso dev può anche essere NULL nel qual caso verranno aggiornati i dati per tutti i filesystem con quote attive, id e addr vengono comunque ignorati.
Q_GETSTATS	Ottiene statistiche ed altre informazioni generali relative al sistema delle quote per il filesystem indicato da dev, richiede che si passi come argomento addr l'indirizzo di una struttura dqstats, mentre i valori di id e dev vengono ignorati; l'operazione è obsoleta e non supportata nei kernel più recenti, che espongono la stessa informazione nei file sotto /proc/self/fs/quota/.

Tabella 4.18: Possibili valori per l'argomento subcmd di QCMD.

privilegi di amministratore.¹⁰⁷ Inoltre gli utenti possono soltanto richiedere i dati relativi alle proprie quote, solo l'amministratore può ottenere i dati di tutti.

Alcune delle operazioni di tab. 4.18 sono alquanto complesse e richiedono un approfondimento maggiore. Le due più rilevanti sono probabilmente Q_GETQUOTA e Q_SETQUOTA, che consentono la gestione dei limiti delle quote. Entrambe fanno riferimento ad una specifica struttura dqblk, la cui definizione è riportata in fig. 4.19,¹⁰⁸ nella quale vengono inseriti i dati relativi alle quote di un singolo utente o gruppo.

La struttura dqblk viene usata sia con Q_GETQUOTA per ottenere i valori correnti dei limiti e dell'occupazione delle risorse, che con Q_SETQUOTA per effettuare modifiche ai limiti. Come si può notare ci sono alcuni campi (in sostanza dqb_curspace, dqb_curinodes, dqb_btime, dqb_itime) che hanno senso solo in lettura, in quanto riportano uno stato non modificabile da quotactl come l'uso corrente di spazio disco ed *inode*, o il tempo che resta nel caso si sia superato un *soft limit*.

Inoltre in caso di modifica di un limite si può voler operare solo su una delle risorse (blocchi o *inode*),¹⁰⁹ per questo la struttura prevede un campo apposito, dqb_valid, il cui scopo è quello di indicare quali sono gli altri campi che devono essere considerati validi.

¹⁰⁷per essere precisi tutte le operazioni indicate come privilegiate in tab. 4.18 richiedono la capacità CAP_SYS_ADMIN.

¹⁰⁸la definizione mostrata è quella usata fino dal kernel 2.4.22, non prenderemo in considerazione le versioni obsolete.

¹⁰⁹non è possibile modificare soltanto uno dei limiti (*hard* o *soft*) occorre sempre rispecificarli entrambi.

```

struct dqblk
{
    u_int64_t dqb_bhardlimit; /* absolute limit on disk quota blocks alloc */
    u_int64_t dqb_bsoftlimit; /* preferred limit on disk quota blocks */
    u_int64_t dqb_curspace; /* current quota block count */
    u_int64_t dqb_ihardlimit; /* maximum # allocated inodes */
    u_int64_t dqb_isoftlimit; /* preferred inode limit */
    u_int64_t dqb_curinodes; /* current # allocated inodes */
    u_int64_t dqb_btime; /* time limit for excessive disk use */
    u_int64_t dqb_itime; /* time limit for excessive files */
    u_int32_t dqb_valid; /* bitmask of QIF_* constants */
};

```

Figura 4.19: La struttura dqblk per i dati delle quote disco.

Questo campo è una maschera binaria che deve essere espressa nei termini di OR aritmetico delle apposite costanti di tab. 4.19, dove si è riportato il significato di ciascuna di esse ed i campi a cui fanno riferimento.

Costante	Descrizione
QIF_BLIMITS	Limiti sui blocchi di spazio disco (dqb_bhardlimit e dqb_bsoftlimit).
QIF_SPACE	Uso corrente dello spazio disco (dqb_curspace).
QIF_ILIMITS	Limiti sugli <i>inode</i> (dqb_ihardlimit e dqb_isoftlimit).
QIF_INODES	Uso corrente degli <i>inode</i> (dqb_curinodes).
QIF_BTIME	Tempo di sfioramento del <i>soft limit</i> sul numero di blocchi (dqb_btime).
QIF_ETIME	Tempo di sfioramento del <i>soft limit</i> sul numero di <i>inode</i> (dqb_etime).
QIF_LIMITS	L'insieme di QIF_BLIMITS e QIF_ILIMITS.
QIF_USAGE	L'insieme di QIF_SPACE e QIF_INODES.
QIF_TIMES	L'insieme di QIF_BTIME e QIF_ETIME.
QIF_ALL	Tutti i precedenti.

Tabella 4.19: Costanti per il campo dqb_valid di dqblk.

In lettura con `Q_SETQUOTA` eventuali valori presenti in `dqblk` vengono comunque ignorati, al momento la funzione sovrascrive tutti i campi che restituisce e li marca come validi in `dqb_valid`. Si possono invece usare `QIF_BLIMITS` o `QIF_ILIMITS` per richiedere di impostare solo la rispettiva tipologia di limiti con `Q_SETQUOTA`. Si tenga presente che il sistema delle quote richiede che l'occupazione di spazio disco sia indicata in termini di blocchi e non di byte, dato che la dimensione dei blocchi dipende da come si è creato il filesystem potrà essere necessario effettuare qualche conversione per avere un valore in byte.¹¹⁰

Come accennato la realizzazione delle quote disco ha visto diverse revisioni, con modifiche sia del formato delle stesse che dei nomi dei file utilizzati. Per questo alcune operazioni di gestione (in particolare `Q_QUOTAON` e `Q_GETFMT`) e possono fare riferimento a queste versioni, che vengono identificate tramite le costanti di tab. 4.20.

Altre due operazioni che necessitano di ulteriori spiegazioni sono `Q_GETINFO` e `Q_SETINFO`, che consentono di ottenere i dati relativi alle impostazioni delle altre proprietà delle quote, che al momento sono solo la durata del *grace time* per i due tipi di limiti. Queste sono due proprietà generali identiche per tutti gli utenti (e i gruppi), per cui viene usata una operazione distinta

¹¹⁰in genere viene usato un default di 1024 byte per blocco, ma quando si hanno file di dimensioni medie maggiori può convenire usare valori più alti per ottenere prestazioni migliori in conseguenza di un minore frazionamento dei dati e di indici più corti.

Identificatore	Descrizione
QFMT_VFS_OLD	Il vecchio (ed obsoleto) formato delle quote.
QFMT_VFS_V0	La versione 0 usata dal VFS di Linux, supporta <i>UID</i> e <i>GID</i> a 32 bit e limiti fino a 2^{42} byte e 2^{32} file.
QFMT_VFS_V1	La versione 1 usata dal VFS di Linux, supporta <i>UID</i> e <i>GID</i> a 32 bit e limiti fino a 2^{64} byte e 2^{64} file.

Tabella 4.20: Valori di identificazione del formato delle quote.

dalle precedenti. Anche in questo caso le due operazioni richiedono l'uso di una apposita struttura `dqinfo`, la cui definizione è riportata in fig. 4.20.

```

struct dqinfo
{
    u_int64_t dqi_bgrace;
    u_int64_t dqi_igrace;
    u_int32_t dqi_flags;
    u_int32_t dqi_valid;
};

```

Figura 4.20: La struttura `dqinfo` per i dati delle quote disco.

Come per `dqblk` anche in questo caso viene usato un campo della struttura, `dqi_valid` come maschera binaria per dichiarare quale degli altri campi sono validi; le costanti usate per comporre questo valore sono riportate in tab. 4.21 dove si è riportato il significato di ciascuna di esse ed i campi a cui fanno riferimento.

Costante	Descrizione
IIF_BGRACE	Il <i>grace period</i> per i blocchi (<code>dqi_bgrace</code>).
IIF_IGRACE	Il <i>grace period</i> per gli <i>inode</i> (<code>dqi_igrace</code>).
IIF_FLAGS	I flag delle quote (<code>dqi_flags</code>) (inusato ?).
IIF_ALL	Tutti i precedenti.

Tabella 4.21: Costanti per il campo `dqi_valid` di `dqinfo`.

Come in precedenza con `Q_GETINFO` tutti i valori vengono letti sovrascrivendo il contenuto di `dqinfo` e marcati come validi in `dqi_valid`. In scrittura con `Q_SETINFO` si può scegliere quali impostare, si tenga presente che i tempi dei campi `dqi_bgrace` e `dqi_igrace` devono essere specificati in secondi.

Come esempi dell'uso di `quotactl` utilizzeremo estratti del codice di un modulo Python usato per fornire una interfaccia diretta a `quotactl` senza dover passare dalla scansione dei risultati di un comando. Il modulo si trova fra i pacchetti Debian messi a disposizione da Truelite Srl, all'indirizzo <http://labs.truelite.it/projects/packages>.¹¹¹

Il primo esempio, riportato in fig. 4.21, riporta il codice della funzione che consente di leggere le quote. La funzione fa uso dell'interfaccia dal C verso Python, che definisce i vari simboli `Py*` (tipi di dato e funzioni). Non staremo ad approfondire i dettagli di questa interfaccia, per la quale esistono numerose trattazioni dettagliate, ci interessa solo esaminare l'uso di `quotactl`.

¹¹¹in particolare il codice C del modulo è nel file `quotamodule.c` visionabile a partire dall'indirizzo indicato nella sezione *Repository*.

```

1 PyObject * get_quota(int who, int id, const char *dev)
2 {
3     struct dqblk dq;
4
5     if (!quotactl(QCMD(Q_GETQUOTA, who), dev, id, (caddr_t) &dq)) {
6         return Py_BuildValue("({s:K,s:(KK),s:K},{s:K,s:(KK),s:K})",
7                               "used", dq.dqb_curspace,
8                               "quota", dq.dqb_bsoftlimit, dq.dqb_bhardlimit,
9                               "grace", dq.dqb_btime,
10                              "used", dq.dqb_curinodes,
11                              "quota", dq.dqb_isoftlimit, dq.dqb_ihardlimit,
12                              "grace", dq.dqb_itype );
13     } else {
14         PyErr_SetFromErrno(PyExc_OSError);
15         return NULL;
16     }
17 }

```

Figura 4.21: Esempio di codice per ottenere i dati delle quote.

In questo caso la funzione prende come argomenti (1) l'intero `who` che indica se si vuole operare sulle quote utente o gruppo, l'identificatore `id` dell'utente o del gruppo scelto, ed il nome del file di dispositivo del filesystem su cui si sono attivate le quote.¹¹² Questi argomenti vengono passati direttamente alla chiamata a `quotactl` (5), a parte `who` che viene abbinato con `QCMD` al comando `Q_GETQUOTA` per ottenere i dati.

La funzione viene eseguita all'interno di un condizionale (5-16) che in caso di successo provvede a costruire (6-12) opportunamente una risposta restituendo tramite la opportuna funzione di interfaccia un oggetto Python contenente i dati della struttura `dqblk` relativi a uso corrente e limiti sia per i blocchi che per gli *inode*. In caso di errore (13-15) si usa un'altra funzione dell'interfaccia per passare il valore di `errno` come eccezione.

```

1 PyObject *set_block_quota(int who, int id, const char *dev, int soft, int hard)
2 {
3     struct dqblk dq;
4
5     dq.dqb_bsoftlimit = soft;
6     dq.dqb_bhardlimit = hard;
7     dq.dqb_valid = QIF_BLIMITS;
8
9     if (!quotactl(QCMD(Q_SETQUOTA, who), dev, id, (caddr_t) &dq)) {
10         Py_RETURN_NONE;
11     } else {
12         PyErr_SetFromErrno(PyExc_OSError);
13         return NULL;
14     }
15 }

```

Figura 4.22: Esempio di codice per impostare i limiti sullo spazio disco.

¹¹²questi vengono passati come argomenti dalle funzioni mappate come interfaccia pubblica del modulo (una per gruppi ed una per gli utenti) che si incaricano di decodificare i dati passati da una chiamata nel codice Python.

Per impostare i limiti sullo spazio disco si potrà usare una seconda funzione, riportata in fig. 4.22, che prende gli stessi argomenti della precedente, con lo stesso significato, a cui si aggiungono i valori per il *soft limit* e l'*hard limit*. In questo caso occorrerà, prima di chiamare `quotactl`, inizializzare opportunamente (5-7) i campi della struttura `dqblk` che si vogliono utilizzare (quelli relativi ai limiti sui blocchi) e specificare gli stessi con `QIF_BLIMITS` in `dq.dqb_valid`.

Fatto questo la chiamata a `quotactl`, stavolta con il comando `Q_SETQUOTA`, viene eseguita come in precedenza all'interno di un condizionale (9-14). In questo caso non essendovi da restituire nessun dato in caso di successo si usa (10) una apposita funzione di uscita, mentre si restituisce come prima una eccezione con il valore di `errno` in caso di errore (12-13).

4.5.4 La gestione dei chroot

Benché non abbia niente a che fare con permessi, utenti e gruppi, la funzione `chroot` viene usata spesso per restringere le capacità di accesso di un programma ad una sezione limitata del filesystem, per cui ne parleremo in questa sezione.

Come accennato in sez. 3.1.3 ogni processo oltre ad una directory di lavoro, ha anche una directory *radice*¹¹³ che, pur essendo di norma corrispondente alla radice dell'albero dei file dell'intero sistema, ha per il processo il significato specifico di directory rispetto alla quale vengono risolti i *pathname* assoluti.¹¹⁴ Il fatto che questo valore sia specificato per ogni processo apre allora la possibilità di modificare le modalità di risoluzione dei *pathname* assoluti da parte di un processo cambiando questa directory, così come si fa coi *pathname* relativi cambiando la directory di lavoro.

Normalmente la directory radice di un processo coincide con la radice generica dell'albero dei file, che è la directory che viene montata direttamente dal kernel all'avvio secondo quanto illustrato in sez. 1.2.2. Questo avviene perché, come visto in sez. 3 la directory radice di un processo viene ereditata dal padre attraverso una `fork` e mantenuta attraverso una `exec`, e siccome tutti i processi derivano da `init`, che ha come radice quella montata dal kernel, questa verrà mantenuta.

In certe situazioni però è utile poter impedire che un processo possa accedere a tutto l'albero dei file iniziale; per far questo si può cambiare la sua directory radice con la funzione di sistema `chroot`, il cui prototipo è:

```
#include <unistd.h>
int chroot(const char *path)
```

Cambia la directory radice del processo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EPERM` non si hanno i privilegi di amministratore.

ed inoltre `EFAULT`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, `EACCES`, `ELOOP`, `EROFS` e `EIO` nel loro significato generico.

La funzione imposta la directory radice del processo a quella specificata da `path` (che ovviamente deve esistere) ed ogni *pathname* assoluto usato dalle funzioni chiamate nel processo sarà risolto a partire da essa, rendendo impossibile accedere alla parte di albero sovrastante. Si ha così quella che viene chiamata una *chroot jail*, in quanto il processo non può più accedere a file al di fuori della sezione di albero in cui è stato *imprigionato*.

¹¹³entrambe sono contenute in due campi (rispettivamente `pwd` e `root`) di `fs_struct`; vedi fig. 3.2.

¹¹⁴cioè quando un processo chiede la risoluzione di un *pathname*, il kernel usa sempre questa directory come punto di partenza.

Solo un processo con i privilegi di amministratore può usare questa funzione,¹¹⁵ e la nuova radice, per quanto detto in sez. 3.1.3, sarà ereditata da tutti i suoi processi figli. Si tenga presente però che la funzione non cambia la directory di lavoro del processo, che potrebbe restare fuori dalla *chroot jail*.

Questo è il motivo per cui la funzione è efficace nel restringere un processo ad un ramo di albero solo se dopo averla eseguita si cedono i privilegi di amministratore. Infatti se per un qualunque motivo il processo resta con la sua directory di lavoro al di fuori della *chroot jail*, potrà accedere a tutto il resto del filesystem usando dei *pathname* relativi, dato che in tal caso è possibile, grazie all'uso di “.”, risalire all'indietro fino alla radice effettiva dell'albero dei file.

Potrebbe sembrare che per risolvere il problema sia sufficiente ricordarsi di eseguire preventivamente anche una `chdir` sulla directory su cui si andrà ad eseguire `chroot`, così da assicurarsi che le directory di lavoro sia all'interno della *chroot jail*. Ma se ad un processo restano i privilegi di amministratore esso potrà comunque portare la sua directory di lavoro fuori dalla *chroot jail* in cui si trova. Basterà infatti eseguire di nuovo `chroot` su una qualunque directory contenuta nell'attuale directory di lavoro perché quest'ultima risulti al di fuori della nuova *chroot jail*. Per questo motivo l'uso di questa funzione non ha molto senso quando un processo di cui si vuole limitare l'accesso necessita comunque dei privilegi di amministratore per le sue normali operazioni.

Nonostante queste limitazioni la funzione risulta utile qualora la si possa applicare correttamente cedendo completamente i privilegi di amministratore una volta eseguita. Ed esempio caso tipico di uso di `chroot` è quello di un server FTP anonimo in si vuole che il server veda solo i file che deve trasferire. In tal caso si esegue una `chroot` sulla directory che contiene i file, che il server dovrà in grado di leggere come utente ordinario, e poi si cedono tutti i privilegi di amministratore. Si tenga presente però che in casi come questo occorrerà fornire all'interno della *chroot jail* un accesso anche a tutti i file (in genere programmi e librerie) di cui il server potrebbe avere bisogno.

¹¹⁵più precisamente se possiede la capacità `CAP_SYS_CHROOT`.

s

Capitolo 5

La gestione dell'I/O su file

Esamineremo in questo capitolo le due interfacce di programmazione che consentono di gestire i dati mantenuti nei file. Cominceremo con quella nativa del sistema, detta dei *file descriptor*, che viene fornita direttamente dalle *system call* e che non prevede funzionalità evolute come la bufferizzazione o funzioni di lettura o scrittura formattata. Esamineremo poi anche l'interfaccia definita dallo standard ANSI C, che viene chiamata dei *file stream* o anche più brevemente degli *stream*. Per entrambe dopo una introduzione alle caratteristiche generali tratteremo le funzioni base per la gestione dell'I/O, lasciando per ultime le caratteristiche più avanzate.

5.1 L'interfaccia dei *file descriptor*

Come visto in sez. 4.1.1 il kernel mette a disposizione tramite il *Virtual File System* una serie di *system call* che consentono di operare sui file in maniera generale. Abbiamo trattato quelle relative alla gestione delle proprietà dei file nel precedente capitolo, vedremo quelle che si applicano al contenuto dei file in questa sezione, iniziando con una breve introduzione sull'architettura dei *file descriptor* per poi trattare le funzioni di base e le modalità con cui consentono di gestire i dati memorizzati sui file.

5.1.1 I *file descriptor*

L'accesso al contenuto dei file viene effettuato, sia pure con differenze nella realizzazione pratica, in maniera sostanzialmente identica in tutte le implementazioni di un sistema unix-like, ricorrendo a quella che viene chiamata l'interfaccia dei *file descriptor*.

Per poter accedere al contenuto di un file occorre creare un canale di comunicazione con il kernel che renda possibile operare su di esso. Questo si fa aprendo il file con la funzione *open* (vedi sez. 5.1.2) che provvederà a localizzare l'*inode* del file e inizializzare i puntatori che rendono disponibili le funzioni che il VFS mette a disposizione (quelle di tab. 4.2). Una volta terminate le operazioni, il file dovrà essere chiuso, e questo chiuderà il canale di comunicazione impedendo ogni ulteriore operazione.

All'interno di ogni processo i file aperti sono identificati da un numero intero non negativo, che viene chiamato appunto *file descriptor*. Quando un file viene aperto la funzione *open* restituisce questo numero, tutte le ulteriori operazioni dovranno essere compiute specificando questo stesso numero come argomento alle varie funzioni dell'interfaccia.

Per capire come funziona il meccanismo occorre spiegare a grandi linee come il kernel gestisce l'interazione fra processi e file. Abbiamo già accennato in sez. 3.1.1 come il kernel

mantenga un elenco di tutti processi nella cosiddetta *process table*. Lo stesso, come accennato in sez. 4.1.1, vale anche per tutti i file aperti, il cui elenco viene mantenuto nella cosiddetta *file table*.

La *process table* è una tabella che contiene una voce per ciascun processo attivo nel sistema. Ciascuna voce è costituita dal puntatore a una struttura di tipo `task_struct` nella quale sono raccolte tutte le informazioni relative al processo, fra queste informazioni c'è anche il puntatore ad una ulteriore struttura di tipo `files_struct`,¹ che contiene le informazioni relative ai file che il processo ha aperto.

La *file table* è una tabella che contiene una voce per ciascun file che è stato aperto nel sistema. Come accennato in sez. 4.1.1 per ogni file aperto viene allocata una struttura `file` e la *file table* è costituita da un elenco di puntatori a ciascuna di queste strutture, che, come illustrato in fig. 4.3, contengono le informazioni necessarie per la gestione dei file, ed in particolare:

- i flag di stato del file nel campo `f_flags`.
- la posizione corrente nel file, il cosiddetto *offset*, nel campo `f_pos`.
- un puntatore alla struttura `inode` che identifica l'*inode* del file.²
- un puntatore `f_op` alla tabella delle funzioni che si possono usare sul file.³

Figura 5.1: Schema della architettura dell'accesso ai file attraverso l'interfaccia dei file descriptor.

In fig. 5.1 si è riportato uno schema semplificato in cui è illustrata questa architettura, ed in cui si sono evidenziate le interrelazioni fra la *file table*, la *process table* e le varie strutture di dati che il kernel mantiene per ciascun file e ciascun processo.

Come si può notare alla fine il collegamento che consente di porre in relazione i file ed i processi è effettuato attraverso i dati mantenuti nella struttura `files_struct`, essa infatti contiene alcune informazioni essenziali come:

- i flag relativi ai file aperti dal processo.
- il numero di file aperti dal processo.
- la *file descriptor table*, una tabella con i puntatori, per ciascun file aperto, alla relativa voce nella *file table*.

In questa infrastruttura un file descriptor non è altro che l'intero positivo che indicizza quest'ultima tabella, e che consente di recuperare il puntatore alla struttura `file` corrispondente al file aperto dal processo a cui era stato assegnato questo indice. Una volta ottenuta grazie al file descriptor la struttura `file` corrispondente al file voluto nella *file table*, il kernel potrà usare le funzioni messe disposizione dal VFS per eseguire sul file tutte le operazioni necessarie.

Il meccanismo dell'apertura dei file prevede che venga sempre fornito il primo file descriptor libero nella tabella, e per questo motivo essi vengono assegnati in successione tutte le volte che si apre un nuovo file, posto che non ne sia stato chiuso nessuno in precedenza.

In tutti i sistemi unix-like esiste una convenzione generale per cui ogni processo si aspetta di avere sempre tre file aperti che, per quanto appena detto, avranno come *file descriptor* i

¹la definizione corrente di questa struttura si trova nel file `include/linux/fdtable.h` dei sorgenti del kernel, quella mostrata in fig. 5.1 è una versione pesantemente semplificata.

²nel kernel 2.4.x si è in realtà passati ad un puntatore ad una struttura `dentry` che punta a sua volta all'*inode* passando per la nuova struttura del VFS.

³quelle della struttura `file_operation`, descritte sommariamente in tab. 4.2.

valori 0, 1 e 2. Il primo file è sempre associato al cosiddetto *standard input*, è cioè il file da cui un processo si aspetta di dover leggere i dati in ingresso. Il secondo file è il cosiddetto *standard output*, cioè quello su cui ci si aspetta di dover scrivere i dati in uscita. Il terzo è lo *standard error*, su cui vengono scritti i dati relativi agli errori.

Benché questa sia alla fine soltanto una convenzione, essa è seguita dalla totalità delle applicazioni, e non aderirvi potrebbe portare a problemi di interoperabilità. Nel caso della shell tutti questi file sono associati al terminale di controllo, e corrispondono quindi alla lettura della tastiera per l'ingresso e alla scrittura sul terminale per l'uscita. Lo standard POSIX.1 provvede, al posto dei valori numerici, tre costanti simboliche, definite in tab. 5.1.

File	Significato
STDIN_FILENO	file descriptor dello <i>standard input</i> .
STDOUT_FILENO	file descriptor dello <i>standard output</i> .
STDERR_FILENO	file descriptor dello <i>standard error</i> .

Tabella 5.1: Costanti definite in `unistd.h` per i file standard.

In fig. 5.1 si è rappresentata una situazione diversa rispetto a quella usuale della shell, in cui tutti e tre questi file fanno riferimento al terminale su cui si opera. Nell'esempio invece viene illustrata la situazione di un programma in cui lo *standard input* è associato ad un file mentre lo *standard output* e lo *standard error* sono associati ad un altro file. Si noti poi come per questi ultimi le strutture *file* nella *file table*, pur essendo distinte, fanno riferimento allo stesso *inode*, dato che il file che è stato aperto lo stesso. Questo è quello che avviene normalmente quando si apre più volte lo stesso file.

Si ritrova quindi anche con le voci della *file table* una situazione analoga di quella delle voci di una directory, con la possibilità di avere più voci che fanno riferimento allo stesso *inode*. L'analogia è in realtà molto stretta perché quando si cancella un file, il kernel verifica anche che non resti nessun riferimento in una qualunque voce della *file table* prima di liberare le risorse ad esso associate e disallocare il relativo *inode*.

Nelle vecchie versioni di Unix (ed anche in Linux fino al kernel 2.0.x) il numero di file aperti era anche soggetto ad un limite massimo dato dalle dimensioni del vettore di puntatori con cui era realizzata la tabella dei file descriptor dentro `files_struct`. Questo limite intrinseco nei kernel più recenti non sussiste più, dato che si è passati da un vettore ad una lista, ma restano i limiti imposti dall'amministratore (vedi sez. 6.1.1).

5.1.2 Apertura, creazione e chiusura di un file

La funzione di sistema `open` è la principale funzione dell'interfaccia di gestione dei file, quella che dato un *pathname* consente di ottenere un file descriptor “*aprendo*” il file corrispondente,⁴ il suo prototipo è:

⁴è `open` che alloca *file*, la inserisce nella *file table* e crea il riferimento nella *files_struct* del processo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags)
int open(const char *pathname, int flags, mode_t mode)
```

Apri un file.

La funzione ritorna il file descriptor in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EEXIST `pathname` esiste e si è specificato `O_CREAT` e `O_EXCL`.

EINTR la funzione era bloccata ed è stata interrotta da un segnale (vedi sez. 7.3.1).

EINVAL si è usato `O_CREAT` indicando un `pathname` con caratteri non supportati dal filesystem sottostante o si è richiesto `O_TMPFILE` senza indicare `O_WRONLY` o `O_RDWR` o si è usato `O_DIRECT` per un filesystem che non lo supporta.

EISDIR `pathname` indica una directory e o si è tentato un accesso che prevede la scrittura o si è usato `O_TMPFILE` con accesso che prevede la scrittura ma il kernel non supporta la funzionalità.

EFBIG il file è troppo grande per essere aperto, in genere dovuto al fatto che si è compilata una applicazione a 32 bit senza abilitare il supporto per le dimensioni a 64 bit; questo è il valore restituito fino al kernel 2.6.23, coi successivi viene restituito `EOVERFLOW` come richiesto da POSIX.1.

ELOOP si sono incontrati troppi collegamenti simbolici nel risolvere `pathname` o si è indicato `O_NOFOLLOW` e `pathname` è un collegamento simbolico (e non si è usato `O_PATH`).

ENODEV `pathname` si riferisce a un file di dispositivo che non esiste.

ENOENT `pathname` non esiste e non si è richiesto `O_CREAT`, o non esiste un suo componente, o si riferisce ad una directory inesistente, si è usato `O_TMPFILE` con accesso che prevede la scrittura ma il kernel non supporta la funzionalità.

ENOTDIR si è specificato `O_DIRECTORY` e `pathname` non è una directory.

ENXIO si sono impostati `O_NONBLOCK` o `O_WRONLY` ed il file è una *fifo* che non viene letta da nessun processo o `pathname` è un file di dispositivo ma il dispositivo è assente.

EPERM si è specificato `O_NOATIME` e non si è né amministratori né proprietari del file.

ETXTBSY si è cercato di accedere in scrittura all'immagine di un programma in esecuzione.

EWouldBlock la funzione si sarebbe bloccata ma si è richiesto `O_NONBLOCK`.

ed inoltre `EACCES`, `EDQUOT`, `EFAULT`, `EMFILE`, `ENAMETOOLONG`, `ENFILE`, `ENOMEM`, `ENOSPC`, `EROFS`, nel loro significato generico.

La funzione apre il file indicato da `pathname` nella modalità indicata da `flags`. Essa può essere invocata in due modi diversi, specificando opzionalmente un terzo argomento `mode`. Qualora il file non esista e venga creato, questo argomento consente di indicare quali permessi dovranno essergli assegnati.⁵ I valori possibili sono gli stessi già visti in sez. 4.4.1 e possono essere specificati come OR binario delle costanti descritte in tab. 4.10. Questi permessi sono comunque filtrati dal valore della *umask* (vedi sez. 4.4.3) del processo.

La funzione restituisce sempre il primo file descriptor libero, una caratteristica che permette di prevedere qual è il valore del file descriptor che si otterrà al ritorno di `open`, e che viene spesso usata dalle applicazioni per sostituire i file corrispondenti ai file standard visti in tab. 5.1. Se ad esempio si chiude lo *standard input* e si apre subito dopo un nuovo file questo diventerà il nuovo *standard input* dato che avrà il file descriptor 0.

Al momento dell'apertura il nuovo file descriptor non è condiviso con nessun altro processo (torneremo sul significato della condivisione dei file descriptor, che in genere si ottiene dopo una *fork*, in sez. 5.2.1) ed è impostato, come accennato in sez. 3.1.6, per restare aperto

⁵questo è possibile solo se si è usato in `flags` uno fra `O_CREATE` e `O_TMPFILE`, in tutti gli altri casi sarà ignorato.

attraverso una `exec`. Inoltre la posizione sul file, il cosiddetto *offset*, è impostata all'inizio del file. Una volta aperto un file si potrà operare su di esso direttamente tramite il file descriptor, e quanto avviene al *pathname* con cui lo si è aperto sarà del tutto ininfluente.

Il comportamento della funzione, e le diverse modalità con cui può essere aperto il file, vengono controllati dall'argomento `flags` il cui valore deve essere indicato come maschera binaria in cui ciascun bit ha un significato specifico. Alcuni di questi bit vanno anche a costituire i cosiddetti *file status flags* (i *flag di stato* del file), che vengono mantenuti nel campo `f_flags` della struttura `file` che abbiamo riportato anche in fig. 5.1).

Ciascun flag viene identificato da una apposita costante, ed il valore di `flags` deve essere specificato come OR aritmetico di queste costanti. Inoltre per evitare problemi di compatibilità con funzionalità che non sono previste o non ancora supportate in versioni meno recenti del kernel, la `open` di Linux ignora i flag che non riconosce, pertanto l'indicazione di un flag inesistente non provoca una condizione di errore.

I vari bit che si possono usare come componenti di `flags` sono divisi in tre gruppi principali. Il primo gruppo è quello dei cosiddetti flag delle *modalità di accesso* (o *access mode flags*), che specificano che tipo di accesso si effettuerà sul file, fra lettura, scrittura e lettura/scrittura. Questa modalità deve essere indicata usando una delle costanti di tab. 5.2.

Flag	Significato
<code>O_RDONLY</code>	Apri il file in sola lettura.
<code>O_WRONLY</code>	Apri il file in sola scrittura.
<code>O_RDWR</code>	Apri il file sia in lettura che in scrittura.

Tabella 5.2: Le tre costanti che identificano le modalità di accesso nell'apertura di un file.

A differenza di tutti gli altri flag che vedremo in seguito, in questo caso non si ha a che fare con singoli bit separati dell'argomento `flags`, ma con un numero composto da due bit. Questo significa ad esempio che la combinazione `O_RDONLY|O_WRONLY` non è affatto equivalente a `O_RDWR`, e non deve essere usata.⁶

La modalità di accesso deve sempre essere specificata quando si apre un file, il valore indicato in `flags` viene salvato nei *file status flags*, e può essere riletto con `fcntl` (vedi sez. 5.2.5), il relativo valore può essere poi ottenuto un AND aritmetico della maschera binaria `O_ACCMODE`, ma non può essere modificato. Nella *glibc* sono definite inoltre `O_READ` come sinonimo di `O_RDONLY` e `O_WRITE` come sinonimo di `O_WRONLY`.⁷

Il secondo gruppo di flag è quello delle *modalità di apertura*,⁸ che permettono di specificare alcune delle caratteristiche del comportamento di `open` nel momento in viene eseguita per aprire un file. Questi flag hanno effetto solo nella chiamata della funzione, non sono memorizzati fra i *file status flags* e non possono essere rilette da `fcntl` (vedi sez. 5.2.5).

⁶in realtà su Linux, dove i valori per le tre costanti di tab. 5.2 sono rispettivamente 0, 1 e 2, il valore 3 viene usato con un significato speciale, ed assolutamente fuori standard, disponibile solo per i file di dispositivo e solo per alcuni driver, in cui si richiede la verifica della capacità di accesso in lettura e scrittura ma viene restituito un file descriptor che non può essere letto o scritto, ma solo usato con una `ioctl` (vedi sez. 5.2.5).

⁷si tratta di definizioni completamente fuori standard, attinenti, insieme a `O_EXEC` che permetterebbe l'apertura di un file per l'esecuzione, ad un non meglio precisato "*GNU system*"; pur essendo equivalenti alle definizioni classiche non è comunque il caso di utilizzarle.

⁸la pagina di manuale di `open` parla di *file creation flags*, ma alcuni di questi flag non hanno nulla a che fare con la creazione dei file, mentre il manuale dalla *glibc* parla di più correttamente di *open-time flags*, dato che si tratta di flag il cui significato ha senso solo al momento dell'apertura del file.

⁹acronimo di *Denial of Service*, si chiamano così attacchi miranti ad impedire un servizio causando una qualche forma di carico eccessivo per il sistema, che resta bloccato nelle risposte all'attacco.

Flag	Significato
O_CREAT	Se il file non esiste verrà creato, con le regole di titolarità del file viste in sez. 4.4.4. Se si imposta questo flag l'argomento <code>mode</code> deve essere sempre specificato.
O_DIRECTORY	Se <code>pathname</code> non è una directory la chiamata fallisce. Questo flag, introdotto con il kernel 2.1.126, è specifico di Linux e serve ad evitare dei possibili <i>DoS</i> ⁹ quando <code>opendir</code> viene chiamata su una <i>fifo</i> o su un dispositivo associato ad una unità a nastri. Non viene usato al di fuori dell'implementazione di <code>opendir</code> , ed è utilizzabile soltanto se si è definita la macro <code>_GNU_SOURCE</code> .
O_EXCL	Deve essere usato in congiunzione con <code>O_CREAT</code> ed in tal caso impone che il file indicato da <code>pathname</code> non sia già esistente (altrimenti causa il fallimento della chiamata con un errore di <code>EEXIST</code>).
O_LARGEFILE	Viene usato sui sistemi a 32 bit per richiedere l'apertura di file molto grandi, la cui dimensione non è rappresentabile con la versione a 32 bit del tipo <code>off_t</code> , utilizzando l'interfaccia alternativa abilitata con la macro <code>_LARGEFILE64_SOURCE</code> . Come illustrato in sez. 1.3.7 è sempre preferibile usare la conversione automatica delle funzioni che si attiva assegnando a 64 la macro <code>_FILE_OFFSET_BITS</code> , e non usare mai questo flag.
O_NOCTTY	Se <code>pathname</code> si riferisce ad un dispositivo di terminale, questo non diventerà il terminale di controllo, anche se il processo non ne ha ancora uno (si veda sez. 8.1.3).
O_NOFOLLOW	Se <code>pathname</code> è un collegamento simbolico la chiamata fallisce. Questa è un'estensione BSD aggiunta in Linux a partire dal kernel 2.1.126, ed utilizzabile soltanto se si è definita la macro <code>_GNU_SOURCE</code> .
O_TMPFILE	Consente di creare un file temporaneo anonimo, non visibile con un <code>pathname</code> sul filesystem, ma leggibile e scrivibile all'interno del processo. Introdotto con il kernel 3.11, è specifico di Linux.
O_TRUNC	Se usato su un file di dati aperto in scrittura, ne tronca la lunghezza a zero; con un terminale o una <i>fifo</i> viene ignorato, negli altri casi il comportamento non è specificato.

Tabella 5.3: Le costanti che identificano le *modalità di apertura* di un file.

Si è riportato in tab. 5.3 l'elenco dei flag delle *modalità di apertura*.¹⁰ Uno di questi, `O_EXCL`, ha senso solo se usato in combinazione a `O_CREAT` quando si vuole creare un nuovo file per assicurarsi che questo non esista di già, e lo si usa spesso per creare i cosiddetti “*file di lock*” (vedi sez. 11.3.2).

Si tenga presente che questa opzione è supportata su NFS solo a partire da NFSv3 e con il kernel 2.6, nelle versioni precedenti la funzionalità viene emulata controllando prima l'esistenza del file per cui usarla per creare un file di lock potrebbe dar luogo a una *race condition*, in tal caso infatti un file potrebbe venir creato fra il controllo la successiva apertura con `O_CREAT`; la cosa si può risolvere comunque creando un file con un nome univoco ed usando la funzione `link` per creare il file di lock, (vedi sez. 11.3.2).

Se si usa `O_EXCL` senza `O_CREAT` il comportamento è indefinito, escluso il caso in cui viene usato con `O_TMPFILE` su cui torneremo più avanti; un'altra eccezione è quello in cui lo si usa da solo su un file di dispositivo, in quel caso se questo è in uso (ad esempio se è relativo ad un filesystem che si è montato) si avrà un errore di `EBUSY`, e pertanto può essere usato in questa modalità per rilevare lo stato del dispositivo.

Nella creazione di un file con `O_CREAT` occorre sempre specificare l'argomento di `mode`, che altrimenti è ignorato. Si tenga presente che indipendentemente dai permessi che si possono assegnare, che in seguito potrebbero non consentire lettura o scrittura, quando il file viene aperto l'accesso viene garantito secondo quanto richiesto con i flag di tab. 5.2. Quando viene

¹⁰la *glibc* definisce anche i due flag `O_SHLOCK`, che aprirebbe il file con uno *shared lock* e `O_EXLOCK` che lo aprirebbe con un *exclusive lock* (vedi sez. 10.1), si tratta di opzioni specifiche di BSD, che non esistono con Linux.

creato un nuovo file `O_CREAT` con tutti e tre i tempi del file di tab. 4.8 vengono impostati al tempo corrente. Se invece si tronca il file con `O_TRUNC` verranno impostati soltanto il *modification time* e lo *status change time*.

Il flag `O_TMPFILE`, introdotto con il kernel 3.11,¹¹ consente di aprire un file temporaneo senza che questo venga associato ad un nome e compaia nel filesystem. In questo caso la funzione restituirà un file descriptor da poter utilizzare per leggere e scrivere dati, ma il contenuto dell'argomento `path` verrà usato solamente per determinare, in base alla directory su cui si verrebbe a trovare il *pathname* indicato, il filesystem all'interno del quale deve essere allocato l'*inode* e lo spazio disco usato dal file descriptor. L'*inode* resterà anonimo e l'unico riferimento esistente sarà quello contenuto nella *file table* del processo che ha chiamato `open`.

Lo scopo principale del flag è quello fornire una modalità atomica, semplice e sicura per applicare la tecnica della creazione di un file temporaneo seguita dalla sua immediata cancellazione con `unlink` per non lasciare rimasugli sul filesystem, di cui è parlato in sez. 4.2.1. Inoltre, dato che il file non compare nel filesystem, si evitano alla radice tutti gli eventuali problemi di *race condition* o *symlink attack* e non ci si deve neanche preoccupare di ottenere un opportuno nome univoco con l'uso delle funzioni di sez. 4.2.6.

Una volta aperto il file vi si potrà leggere o scrivere a seconda che siano utilizzati `O_RDWR` o `O_WRONLY`, mentre l'uso di `O_RDONLY` non è consentito, non avendo molto senso ottenere un file descriptor su un file che nasce vuoto per cui non si potrà comunque leggere nulla. L'unico altro flag che può essere utilizzato insieme a `O_TMPFILE` è `O_EXCL`, che in questo caso assume però un significato diverso da quello ordinario, dato che in questo caso il file associato al file descriptor non esiste comunque.

L'uso di `O_EXCL` attiene infatti all'altro possibile impiego di `O_TMPFILE` oltre a quello citato della creazione sicura di un file temporaneo come sostituto sicuro di `tmpfile`: la possibilità di creare un contenuto iniziale per un file ed impostarne permessi, proprietario e attributi estesi con `fchmod`, `fchown` e `fsetxattr`, senza possibilità di *race condition* ed interferenze esterne, per poi far apparire il tutto sul filesystem in un secondo tempo utilizzando `linkat` sul file descriptor (torneremo su questo in sez. 5.2.4) per dargli un nome. Questa operazione però non sarà possibile se si è usato `O_EXCL`, che in questo caso viene ad assumere il significato di escludere la possibilità di far esistere il file anche in un secondo tempo.

Il terzo gruppo è quello dei flag delle *modalità di operazione*, riportati in tab. 5.4, che permettono di specificare varie caratteristiche del comportamento delle operazioni di I/O che verranno eseguite sul file o le modalità in cui lo si potrà utilizzare. Tutti questi, tranne `O_CLOEXEC`, che viene mantenuto per ogni singolo file descriptor, vengono salvati nel campo `f_flags` della struttura `file` insieme al valore della *modalità di accesso*, andando far parte dei *file status flags*. Il loro valore viene impostato alla chiamata di `open`, ma possono venire riletti in un secondo tempo con `fcntl`, inoltre alcuni di essi possono anche essere modificati tramite questa funzione, con conseguente effetto sulle caratteristiche operative che controllano (torneremo sull'argomento in sez. 5.2.5).

Il flag `O_ASYNC` (che, per compatibilità con BSD, si può indicare anche con la costante `FASYNC`) è definito come possibile valore per `open`, ma per un bug dell'implementazione,¹² non solo non attiva il comportamento citato, ma se usato richiede di essere esplicitamente disattivato prima di essere attivato in maniera effettiva con l'uso di `fcntl`. Per questo motivo, non essendovi nessuna necessità specifica di definirlo in fase di apertura del file, è sempre opportuno attivarlo in un secondo tempo con `fcntl` (vedi sez. 5.2.5).

¹¹inizialmente solo su alcuni filesystem (i vari *extN*, *Minix*, *UDF*, *shmem*) poi progressivamente esteso ad altri (*XFS* con il 3.15, *Btrfs* e *F2FS* con il 3.16, *ubifs* con il 4.9).

¹²segnalato come ancora presente nella pagina di manuale, almeno fino al novembre 2018.

Flag	Significato
O_APPEND	Il file viene aperto in <i>append mode</i> . La posizione sul file (vedi sez. 5.1.3) viene sempre mantenuta sulla sua coda, per cui quanto si scrive viene sempre aggiunto al contenuto precedente. Con NFS questa funzionalità non è supportata e viene emulata, per questo possono verificarsi <i>race condition</i> con una sovrapposizione dei dati se più di un processo scrive allo stesso tempo.
O_ASYNC	Apri il file per l'I/O in modalità asincrona (vedi sez. 10.3.1). Quando è impostato viene generato il segnale SIGIO tutte le volte che il file è pronto per le operazioni di lettura o scrittura. Questo flag si può usare solo terminali, pseudo-terminali e socket e, a partire dal kernel 2.6, anche sulle <i>fifo</i> . Per un bug dell'implementazione non è opportuno usarlo in fase di apertura del file, deve invece essere attivato successivamente con <code>fcntl</code> .
O_CLOEXEC	Attiva la modalità di <i>close-on-exec</i> (vedi sez. 3.1.6) sul file. Il flag è previsto dallo standard POSIX.1-2008, ed è stato introdotto con il kernel 2.6.23 per evitare una <i>race condition</i> che si potrebbe verificare con i <i>thread</i> fra l'apertura del file e l'impostazione della suddetta modalità con <code>fcntl</code> (vedi sez. 5.2.5).
O_DIRECT	Esegue l'I/O direttamente dalla memoria in <i>user space</i> in maniera sincrona, in modo da scavalcare i meccanismi di bufferizzazione del kernel. Introdotto con il kernel 2.4.10 ed utilizzabile soltanto se si è definita la macro <code>_GNU_SOURCE</code> .
O_NOATIME	Blocca l'aggiornamento dei tempi di accesso dei file (vedi sez. 4.3.4). Per molti filesystem questa funzionalità non è disponibile per il singolo file ma come opzione generale da specificare in fase di montaggio. Introdotto con il kernel 2.6.8 ed utilizzabile soltanto se si è definita la macro <code>_GNU_SOURCE</code> .
O_NONBLOCK	Apri il file in <i>modalità non bloccante</i> per le operazioni di I/O (vedi sez. 10.2.1). Questo significa il fallimento delle successive operazioni di lettura o scrittura qualora il file non sia pronto per la loro esecuzione immediata, invece del blocco delle stesse in attesa di una successiva possibilità di esecuzione come avviene normalmente. Questa modalità ha senso solo per le <i>fifo</i> , vedi sez. 11.1.4, o quando si vuole aprire un file di dispositivo per eseguire una <code>ioctl</code> (vedi sez. 5.2.5).
O_NDELAY	In Linux è un sinonimo di <code>O_NONBLOCK</code> , ma origina da SVr4, dove però causava il ritorno da una <code>read</code> con un valore nullo e non con un errore, questo introduce un'ambiguità, dato che come vedremo in sez. 5.1.4 il ritorno di un valore nullo da parte di <code>read</code> ha il significato di una <i>end-of-file</i> .
O_PATH	Ottiene un file descriptor io cui uso è limitato all'indicare una posizione sul filesystem o eseguire operazioni che operano solo a livello del file descriptor (e non di accesso al contenuto del file). Introdotto con il kernel 2.6.39, è specifico di Linux.
O_SYNC	Apri il file per l'input/output sincrono. Ogni scrittura si bloccherà fino alla conferma dell'arrivo di tutti i dati e di tutti i metadati sull'hardware sottostante (in questo significato solo dal kernel 2.6.33).
O_DSYNC	Apri il file per l'input/output sincrono. Ogni scrittura di dati si bloccherà fino alla conferma dell'arrivo degli stessi e della parte di metadati ad essi relativa sull'hardware sottostante (in questo significato solo dal kernel 2.6.33).

Tabella 5.4: Le costanti che identificano le *modalità di operazione* di un file.

Il flag `O_DIRECT` non è previsto da nessuno standard, anche se è presente in alcuni kernel unix-like.¹³ Per i kernel della serie 2.4 si deve garantire che i buffer in *user space* da cui si effettua il trasferimento diretto dei dati siano allineati alle dimensioni dei blocchi del filesystem. Con il kernel 2.6 in genere basta che siano allineati a multipli di 512 byte, ma le restrizioni possono variare a seconda del filesystem, ed inoltre su alcuni filesystem questo flag può non essere supportato, nel qual caso si avrà un errore di `EINVAL`.

Lo scopo di `O_DIRECT` è consentire un completo controllo sulla bufferizzazione dei propri dati per quelle applicazioni (in genere database) che hanno esigenze specifiche che non ven-

¹³il flag è stato introdotto dalla SGI in IRIX, ma è presente senza limiti di allineamento dei buffer anche in FreeBSD.

gono soddisfatte nella maniera più efficiente dalla politica generica utilizzata dal kernel. In genere l'uso di questo flag peggiora le prestazioni tranne quando le applicazioni sono in grado di ottimizzare la propria bufferizzazione in maniera adeguata. Se lo si usa si deve avere cura di non mescolare questo tipo di accesso con quello ordinario, in quante le esigenze di mantenere coerenti i dati porterebbero ad un peggioramento delle prestazioni. Lo stesso dicasi per l'interazione con eventuale mappatura in memoria del file (vedi sez. 10.4.1).

Si tenga presente infine che anche se l'uso di `O_DIRECT` comporta una scrittura sincrona dei dati dei buffer in *user space*, questo non è completamente equivalente all'uso di `O_SYNC` che garantisce anche sulla scrittura sincrona dei metadati associati alla scrittura dei dati del file.¹⁴ Per questo in genere se si usa `O_DIRECT` è opportuno richiedere anche `O_SYNC`.

Si tenga presente infine che la implementazione di `O_SYNC` di Linux differisce da quanto previsto dallo standard POSIX.1 che prevede, oltre a questo flag che dovrebbe indicare la sincronizzazione completa di tutti i dati e di tutti i metadati, altri due flag `O_DSYNC` e `O_RSYNC`.

Il primo dei due richiede la scrittura sincrona di tutti i dati del file e dei metadati che ne consentono l'immediata riletture, ma non di tutti i metadati, per evitare la perdita di prestazioni relativa alla sincronizzazione di informazioni ausiliarie come i tempi dei file. Il secondo, da usare in combinazione con `O_SYNC` o `O_DSYNC` ne sospende l'effetto, consentendo al kernel di bufferizzare le scritture, ma soltanto finché non avviene una lettura, in quel caso i dati ed i metadati dovranno essere sincronizzati immediatamente (secondo le modalità indicate da `O_SYNC` e `O_DSYNC`) e la lettura verrà bloccata fintanto che detta sincronizzazione non sia completata.

Nel caso di Linux, fino al kernel 2.6.33, esisteva solo `O_SYNC`, ma con il comportamento previsto dallo standard per `O_DSYNC`, e sia questo che `O_RSYNC` erano definiti (fin dal kernel 2.1.130) come sinonimi di `O_SYNC`. Con il kernel 2.6.33 il significato di `O_SYNC` è diventato quello dello standard, ma gli è stato assegnato un valore diverso, mantenendo quello originario, con il comportamento corrispondente, per `O_DSYNC` in modo che applicazioni compilate con versioni precedenti delle librerie e del kernel non trovassero un comportamento diverso. Inoltre il nuovo `O_SYNC` è stato definito in maniera opportuna in modo che su versioni del kernel precedenti la 2.6.33 torni a corrispondere al valore di `O_DSYNC`.

Il flag `O_PATH`, introdotto con il kernel 2.6.39, viene usato per limitare l'uso del file descriptor restituito da `open` o all'identificazione di una posizione sul filesystem (ad uso delle *at-functions* che tratteremo in sez. 5.2.4) o alle operazioni che riguardano il file descriptor in quanto tale, senza consentire operazioni sul file; in sostanza se si apre un file con `O_PATH` si potrà soltanto:

- usare il file descriptor come indicatore della directory di partenza con una delle *at-functions* (vedi sez. 5.2.4);
- cambiare directory di lavoro con `fchdir` se il file descriptor fa riferimento a una directory (dal kernel 3.5);
- usare le funzioni che duplicano il file descriptor (vedi sez. 5.2.2);
- passare il file descriptor ad un altro processo usando un socket `PF_UNIX` (vedi sez. 17.2)
- ottenere le informazioni relative al file con `fstat` (dal kernel 3.6) o al filesystem con `fstatfs` (dal kernel 3.12);
- ottenere il valore dei *file descriptor flags* (fra cui comparirà anche lo stesso `O_PATH`) o impostare o leggere i *file status flags* con `fcntl` (rispettivamente con le operazioni `F_GETFL`, `F_SETFD` e `F_GETFD`, vedi sez. 5.2.5).
- chiudere il file con `close`.

¹⁴la situazione si complica ulteriormente per NFS, in cui l'uso del flag disabilita la bufferizzazione solo dal lato del client, e può causare problemi di prestazioni.

In realtà usando `O_PATH` il file non viene effettivamente aperto, per cui ogni tentativo di usare il file descriptor così ottenuto con funzioni che operano effettivamente sul file (come ad esempio `read`, `write`, `fchown`, `fchmod`, `ioctl`, ecc.) fallirà con un errore di `EBADF`, come se questo non fosse un file descriptor valido. Per questo motivo usando questo flag non è necessario avere nessun permesso per aprire un file, neanche quello di lettura (ma occorre ovviamente avere il permesso di esecuzione per le directory sovrastanti).

Questo consente di usare il file descriptor con funzioni che non richiedono permessi sul file, come `fstat`, laddove un'apertura con `O_RDONLY` sarebbe fallita. I permessi verranno eventualmente controllati, se necessario, nelle operazioni seguenti, ad esempio per usare `fchdir` con il file descriptor (se questo fa riferimento ad una directory) occorrerà avere il permesso di esecuzione.

Se si usa `O_PATH` tutti gli altri flag eccettuati `O_CLOEXEC`, `O_DIRECTORY` e `O_NOFOLLOW` verranno ignorati. I primi due mantengono il loro significato usuale, mentre `O_NOFOLLOW` fa sì che se il file indicato è un link simbolico venga aperto quest'ultimo (cambiando quindi il comportamento ordinario che prova il fallimento della chiamata), così da poter usare il file descriptor ottenuto per le funzioni `fchownat`, `fstatat`, `linkat` e `readlinkat` che ne supportano l'uso come primo argomento (torneremo su questo in sez. 5.2.4).

Nelle prime versioni di Unix i valori di flag specificabili per `open` erano solo quelli relativi alle modalità di accesso del file. Per questo motivo per creare un nuovo file c'era una *system call* apposita, `creat`, nel caso di Linux questo non è più necessario ma la funzione è definita ugualmente; il suo prototipo è:

<pre>#include <fcntl.h> int creat(const char *pathname, mode_t mode)</pre>	Crea un nuovo file vuoto.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà gli stessi valori che si otterrebbero con <code>open</code> .	

La funzione crea un nuovo file vuoto, con i permessi specificati dall'argomento `mode`. È del tutto equivalente a `open(filedes, O_CREAT|O_WRONLY|O_TRUNC, mode)` e resta solo per compatibilità con i vecchi programmi.

Una volta che l'accesso ad un file non sia più necessario la funzione di sistema `close` permette di “chiuderlo”, in questo modo il file non sarà più accessibile ed il relativo file descriptor ritornerà disponibile; il suo prototipo è:

<pre>#include <unistd.h> int close(int fd)</pre>	Chiude un file.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori: <code>EBADF</code> <code>fd</code> non è un descrittore valido. <code>EINTR</code> la funzione è stata interrotta da un segnale. ed inoltre <code>EIO</code> nel suo significato generico.	

La funzione chiude il file descriptor `fd`. La chiusura rilascia ogni eventuale blocco (il *file locking* è trattato in sez. 10.1) che il processo poteva avere acquisito su di esso. Se `fd` è l'ultimo riferimento (di eventuali copie, vedi sez. 5.2.1 e 5.2.2) ad un file aperto, tutte le risorse nella *file table* vengono rilasciate. Infine se il file descriptor era l'ultimo riferimento ad un file su disco quest'ultimo viene cancellato.

Si ricordi che quando un processo termina tutti i suoi file descriptor vengono automaticamente chiusi, molti programmi sfruttano questa caratteristica e non usano esplicitamente `close`. In genere comunque chiudere un file senza controllare lo stato di uscita di `close` un

è errore; molti filesystem infatti implementano la tecnica del cosiddetto *write-behind*, per cui una `write` può avere successo anche se i dati non sono stati effettivamente scritti su disco. In questo caso un eventuale errore di I/O avvenuto in un secondo tempo potrebbe sfuggire, mentre verrebbe riportato alla chiusura esplicita del file. Per questo motivo non effettuare il controllo può portare ad una perdita di dati inavvertita.¹⁵

In ogni caso una `close` andata a buon fine non garantisce che i dati siano stati effettivamente scritti su disco, perché il kernel può decidere di ottimizzare l'accesso a disco ritardandone la scrittura. L'uso della funzione `sync` (vedi sez. 5.2.3) effettua esplicitamente lo scarico dei dati, ma anche in questo caso resta l'incertezza dovuta al comportamento dell'hardware, che a sua volta può introdurre ottimizzazioni dell'accesso al disco che ritardano la scrittura dei dati. Da questo deriva l'abitudine di alcuni sistemisti di ripetere tre volte il comando omonimo prima di eseguire lo shutdown di una macchina.

Si tenga comunque presente che ripetere la chiusura in caso di fallimento non è opportuno, una volta chiamata `close` il file descriptor viene comunque rilasciato, indipendentemente dalla presenza di errori, e se la riesecuzione non comporta teoricamente problemi (se non la sua inutilità) se fatta all'interno di un processo singolo, nel caso si usino i *thread* si potrebbe chiudere un file descriptor aperto nel contempo da un altro *thread*.

5.1.3 La gestione della posizione nel file

Come già accennato in sez. 5.1.1 a ciascun file aperto è associata una *posizione corrente nel file* (il cosiddetto *file offset*, mantenuto nel campo `f_pos` di `file`) espressa da un numero intero positivo che esprime il numero di byte dall'inizio del file. Tutte le operazioni di lettura e scrittura avvengono a partire da questa posizione che viene automaticamente spostata in avanti del numero di byte letti o scritti.

In genere, a meno di non avere richiesto la modalità di scrittura in *append* (vedi sez. 5.1.5) con `O_APPEND`, questa posizione viene impostata a zero all'apertura del file. È possibile impostarla ad un valore qualsiasi con la funzione di sistema `lseek`, il cui prototipo è:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence)
```

Imposta la posizione sul file.

La funzione ritorna il valore della posizione sul file in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `whence` non è un valore valido.

`EOVERFLOW` `offset` non può essere rappresentato nel tipo `off_t`.

`ESPIPE` `fd` è una *pipe*, un socket o una *fifo*.

ed inoltre `EBADF` nel suo significato generico.

La funzione imposta la nuova posizione sul file usando il valore indicato da `offset`, che viene sommato al riferimento dato dall'argomento `whence`, che deve essere indicato con una delle costanti riportate in tab. 5.5.¹⁶ Si tenga presente che la chiamata a `lseek` non causa nessun accesso al file, si limita a modificare la posizione corrente (cioè il campo `f_pos` della struttura `file`, vedi fig. 5.1). Dato che la funzione ritorna la nuova posizione, usando il valore zero per `offset` si può riottenere la posizione corrente nel file con `lseek(fd, 0, SEEK_CUR)`.

Si tenga presente inoltre che usare `SEEK_END` non assicura affatto che la successiva scrittura avvenga alla fine del file, infatti se questo è stato aperto anche da un altro processo che vi

¹⁵in Linux questo comportamento è stato osservato con NFS e le quote su disco.

¹⁶per compatibilità con alcune vecchie notazioni questi valori possono essere rimpiazzati rispettivamente con 0, 1 e 2 o con `L_SET`, `L_INCR` e `L_XTND`.

Costante	Significato
SEEK_SET	Si fa riferimento all'inizio del file: il valore, che deve essere positivo, di <code>offset</code> indica direttamente la nuova posizione corrente.
SEEK_CUR	Si fa riferimento alla posizione corrente del file: ad essa viene sommato <code>offset</code> , che può essere negativo e positivo, per ottenere la nuova posizione corrente.
SEEK_END	Si fa riferimento alla fine del file: alle dimensioni del file viene sommato <code>offset</code> , che può essere negativo e positivo, per ottenere la nuova posizione corrente.
SEEK_DATA	Sposta la posizione nel file sull'inizio del primo blocco di dati dopo un <i>hole</i> che segue (o coincide) con la posizione indicata da <code>offset</code> (dal kernel 3.1).
SEEK_HOLE	Sposta la posizione sul file all'inizio del primo <i>hole</i> nel file che segue o inizia con <code>offset</code> , oppure si porta su <code>offset</code> se questo è all'interno di un <i>hole</i> , oppure si porta alla fine del file se non ci sono <i>hole</i> dopo <code>offset</code> (dal kernel 3.1).

Tabella 5.5: Possibili valori per l'argomento `whence` di `lseek`.

ha scritto, la fine del file può essersi spostata, ma noi scriveremo alla posizione impostata in precedenza (questa è una potenziale sorgente di *race condition*, vedi sez. 5.2.1).

Non tutti i file supportano la capacità di eseguire una `lseek`, in questo caso la funzione ritorna l'errore `ESPIPE`. Questo, oltre che per i tre casi citati nel prototipo, vale anche per tutti quei dispositivi che non supportano questa funzione, come ad esempio per i file di terminale.¹⁷ Lo standard POSIX però non specifica niente in proposito. Inoltre alcuni file speciali, ad esempio `/dev/null`, non causano un errore ma restituiscono un valore indefinito.

Infine si tenga presente che, come accennato in sez. 4.3.3, con `lseek` è possibile impostare una posizione anche oltre la corrente fine del file. In tal caso alla successiva scrittura il file sarà esteso a partire da detta posizione, con la creazione di quello che viene chiamato un “buco” (in gergo *hole*) nel file. Il nome deriva dal fatto che nonostante la dimensione del file sia cresciuta in seguito alla scrittura effettuata, lo spazio vuoto fra la precedente fine del file e la nuova parte, scritta dopo lo spostamento, non corrisponde ad una allocazione effettiva di spazio su disco, che sarebbe inutile dato che quella zona è effettivamente vuota.

Questa è una delle caratteristiche specifiche della gestione dei file di un sistema unix-like e quando si ha questa situazione si dice che il file in questione è uno *sparse file*. In sostanza, se si ricorda la struttura di un filesystem illustrata in fig. 4.5, quello che accade è che nell'*inode* del file viene segnata l'allocazione di un blocco di dati a partire dalla nuova posizione, ma non viene allocato nulla per le posizioni intermedie. In caso di lettura sequenziale del contenuto del file il kernel si accorgerà della presenza del buco, e restituirà degli zeri come contenuto di quella parte del file.

Questa funzionalità comporta una delle caratteristiche della gestione dei file su Unix che spesso genera più confusione in chi non la conosce, per cui sommando le dimensioni dei file si può ottenere, se si hanno molti *sparse file*, un totale anche maggiore della capacità del proprio disco e comunque maggiore della dimensione che riporta un comando come `du`, che calcola lo spazio disco occupato in base al numero dei blocchi effettivamente allocati per il file.

Tutto ciò avviene proprio perché in un sistema unix-like la dimensione di un file è una caratteristica del tutto indipendente dalla quantità di spazio disco effettivamente allocato, e viene registrata sull'*inode* come le altre proprietà del file. La dimensione viene aggiornata automaticamente quando si estende un file scrivendoci, e viene riportata dal campo `st_size` di una struttura `stat` quando si effettua la chiamata ad una delle funzioni `*stat` viste in sez. 4.3.1.

Questo comporta che la dimensione di un file, fintanto che lo si è scritto sequenzialmente, sarà corrispondente alla quantità di spazio disco da esso occupato, ma possono esistere dei

¹⁷altri sistemi, usando `SEEK_SET`, in questo caso ritornano il numero di caratteri che vi sono stati scritti.

casi, come questo in cui ci si sposta in una posizione oltre la fine corrente del file, o come quello accennato in sez. 4.3.3 in cui si estende la dimensione di un file con una `truncate`, in cui si modifica soltanto il valore della dimensione di `st_size` senza allocare spazio su disco. Così è possibile creare inizialmente file di dimensioni anche molto grandi, senza dover occupare da subito dello spazio disco che in realtà sarebbe inutilizzato.

A partire dal kernel 3.1, riprendendo una interfaccia adottata su Solaris, sono state aggiunti due nuovi valori per l'argomento `whence`, riportati nella seconda sezione di tab. 5.5, che consentono di riconoscere la presenza di *hole* all'interno dei file ad uso di quelle applicazioni (come i programmi di backup) che possono salvare spazio disco nella copia degli *sparse file*. Una applicazione può così determinare la presenza di un *hole* usando `SEEK_HOLE` all'inizio del file e determinare poi l'inizio della successiva sezione di dati usando `SEEK_DATA`. Per compatibilità con i filesystem che non supportano questa funzionalità è previsto comunque che in tal caso `SEEK_HOLE` riporti sempre la fine del file e `SEEK_DATA` il valore di `offset`.

Inoltre la decisione di come riportare (o di non riportare) la presenza di un buco in un file è lasciata all'implementazione del filesystem, dato che oltre a quelle classiche appena esposte esistono vari motivi per cui una sezione di un file può non contenere dati ed essere riportata come tale (ad esempio può essere stata preallocata con `fallocate`, vedi sez. 10.4.4). Questo significa che l'uso di questi nuovi valori non garantisce la mappatura della effettiva allocazione dello spazio disco di un file, per il quale esiste una specifica operazione di controllo (vedi sez. 5.2.5).

5.1.4 Le funzioni per la lettura di un file

Una volta che un file è stato aperto (con il permesso in lettura) si possono leggere i dati che contiene utilizzando la funzione di sistema `read`, il cui prototipo è:

```
#include <unistd.h>
ssize_t read(int fd, void * buf, size_t count)
```

Legge i dati da un file.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>EAGAIN</code>	la funzione non ha nessun dato da restituire e si è aperto il file con <code>O_NONBLOCK</code> .
<code>EINTR</code>	la funzione è stata interrotta da un segnale.
<code>EINVAL</code>	<code>fd</code> è associato ad un oggetto non leggibile, o lo si è ottenuto da <code>timerfd_create</code> (vedi sez. 10.2.5) e si è usato un valore sbagliato per <code>size</code> o si è usato <code>O_DIRECT</code> ed il buffer non è allineato.
<code>EIO</code>	si è tentata la lettura dal terminale di controllo essendo in background ignorando o bloccando <code>SIGTTIN</code> (vedi sez. 8.2.1) o per errori di basso livello sul supporto.

ed inoltre `EBADF`, `EFAULT` e `EISDIR`, nel loro significato generico.

La funzione tenta di leggere `count` byte dal file `fd` a partire dalla posizione corrente, scrivendoli nel buffer `buf`.¹⁸ Dopo la lettura la posizione sul file è spostata automaticamente in avanti del numero di byte letti. Se `count` è zero la funzione restituisce zero senza nessun altro risultato. Inoltre che non è detto che la funzione `read` restituisca il numero di byte richiesto, ci sono infatti varie ragioni per cui la funzione può restituire un numero di byte inferiore: questo è un comportamento normale, e non un errore, che bisogna sempre tenere presente.

La prima e più ovvia di queste ragioni è che si è chiesto di leggere più byte di quanto il file ne contenga. In questo caso il file viene letto fino alla sua fine, e la funzione ritorna

¹⁸fino ad un massimo di `0x7ffff000` byte, indipendentemente che l'architettura sia a 32 o 64 bit.

regolarmente il numero di byte letti effettivamente. Raggiunta la fine del file, alla ripetizione di un'operazione di lettura, otterremmo il ritorno immediato di `read` con uno zero. La condizione di raggiungimento della fine del file non è un errore, e viene segnalata appunto da un valore di ritorno di `read` nullo. Ripetere ulteriormente la lettura non avrebbe nessun effetto se non quello di continuare a ricevere zero come valore di ritorno.

Con i *file regolari* questa è l'unica situazione in cui si può avere un numero di byte letti inferiore a quello richiesto, ma questo non è vero quando si legge da un terminale, da una *fifo* o da una *pipe*. In tal caso infatti, se non ci sono dati in ingresso, la `read` si blocca (a meno di non aver selezionato la modalità non bloccante, vedi sez. 10.2.1) e ritorna solo quando ne arrivano; se il numero di byte richiesti eccede quelli disponibili la funzione ritorna comunque, ma con un numero di byte inferiore a quelli richiesti.

Lo stesso comportamento avviene caso di lettura dalla rete (cioè su un socket, come vedremo in sez. 15.3.1), o per la lettura da certi file di dispositivo, come le unità a nastro, che restituiscono sempre i dati ad un singolo blocco alla volta, o come le linee seriali, che restituiscono solo i dati ricevuti fino al momento della lettura, o i terminali, per i quali si applicano anche ulteriori condizioni che approfondiremo in sez. 8.2.

Infine anche le due condizioni segnalate dagli errori `EINTR` ed `EAGAIN` non sono propriamente degli errori. La prima si verifica quando la `read` è bloccata in attesa di dati in ingresso e viene interrotta da un segnale. In tal caso l'azione da intraprendere è quella di rieseguire la funzione, torneremo in dettaglio sull'argomento in sez. 7.3.1. La seconda si verifica quando il file è aperto in modalità non bloccante (con `O_NONBLOCK`) e non ci sono dati in ingresso: la funzione allora ritorna immediatamente con un errore `EAGAIN`¹⁹ che indica soltanto che non essendoci al momento dati disponibili occorre provare a ripetere la lettura in un secondo tempo, torneremo sull'argomento in sez. 10.2.1.

La funzione `read` è una delle *system call* fondamentali, esistenti fin dagli albori di Unix, ma nella seconda versione delle *Single Unix Specification*²⁰ (quello che viene chiamato normalmente Unix98, vedi sez. 1.3.6) è stata introdotta la definizione di un'altra funzione di sistema, `pread`, il cui prototipo è:

```
#include <unistd.h>
ssize_t pread(int fd, void * buf, size_t count, off_t offset)
                                                    Legge a partire da una posizione sul file.
```

La funzione ritorna il numero di byte letti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà i valori già visti per `read` e `lseek`.

La funzione prende esattamente gli stessi argomenti di `read` con lo stesso significato, a cui si aggiunge l'argomento `offset` che indica una posizione sul file a partire dalla quale verranno letti i `count` byte. Identico è il comportamento ed il valore di ritorno, ma la posizione corrente sul file resterà invariata. Il valore di `offset` fa sempre riferimento all'inizio del file.

L'uso di `pread` è equivalente all'esecuzione di una `lseek` alla posizione indicata da `offset` seguita da una `read`, seguita da un'altra `lseek` che riporti al valore iniziale della posizione corrente sul file, ma permette di eseguire l'operazione atomicamente. Questo può essere importante quando la posizione sul file viene condivisa da processi diversi (vedi sez. 5.2.1) ed è particolarmente utile in caso di programmazione *multi-thread* (vedi sez. 12) quando all'inter-

¹⁹in BSD si usa per questo errore la costante `EWOULDBLOCK`, in Linux, con la *glibc*, questa è sinonima di `EAGAIN`, ma se si vuole essere completamente portabili occorre verificare entrambi i valori, dato che POSIX.1-2001 non richiede che siano coincidenti.

²⁰questa funzione, e l'analoga `pwrite` sono state aggiunte nel kernel 2.1.60, il supporto nella *glibc*, compresa l'emulazione per i vecchi kernel che non hanno la *system call*, è stato aggiunto con la versione 2.1, in versioni precedenti sia del kernel che delle librerie la funzione non è disponibile.

no di un processo si vuole che le operazioni di un *thread* non possano essere influenzata da eventuali variazioni della posizione sul file effettuate da altri *thread*.

La funzione `pread` è disponibile anche in Linux, però diventa accessibile solo attivando il supporto delle estensioni previste dalle *Single Unix Specification* con un valore della macro `_XOPEN_SOURCE` maggiore o uguale a 500 o a partire dalla *glibc* 2.12 con un valore della macro `_POSIX_C_SOURCE` maggiore o uguale al valore 200809L. Si ricordi di definire queste macro prima dell'inclusione del file di dichiarazione `unistd.h`.

5.1.5 Le funzioni per la scrittura di un file

Una volta che un file è stato aperto (con il permesso in scrittura) si può scrivere su di esso utilizzando la funzione di sistema `write`, il cui prototipo è:

```
#include <unistd.h>
ssize_t write(int fd, void * buf, size_t count)
```

Scrive i dati su un file.

La funzione ritorna il numero di byte scritti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` ci si sarebbe bloccati, ma il file era aperto in modalità `O_NONBLOCK`.

`EDESTADDRREQ` si è eseguita una scrittura su un socket di tipo *datagram* (vedi sez. 14.1.4) senza aver prima connesso il corrispondente con `connect` (vedi sez. 17.1.2).

`EFBIG` si è cercato di scrivere oltre la dimensione massima consentita dal filesystem o il limite per le dimensioni dei file del processo o su una posizione oltre il massimo consentito.

`EINTR` si è stati interrotti da un segnale prima di aver potuto scrivere qualsiasi dato.

`EINVAL` `fd` è connesso ad un oggetto che non consente la scrittura o si è usato `O_DIRECT` ed il buffer non è allineato.

`EPERM` la scrittura è proibita da un *file seal* (vedi sez. 5.2.5).

`EPIPE` `fd` è connesso ad una *pipe* il cui altro capo è chiuso in lettura; in questo caso viene anche generato il segnale `SIGPIPE`, se questo viene gestito (o bloccato o ignorato) la funzione ritorna questo errore.

ed inoltre `EBADF`, `EDQUOT`, `EFAULT`, `EIO`, `EISDIR`, `ENOSPC` nel loro significato generico.

Come nel caso di `read` la funzione tenta di scrivere `count` byte a partire dalla posizione corrente nel file e sposta automaticamente la posizione in avanti del numero di byte scritti. Se il file è aperto in modalità *append* con `O_APPEND` i dati vengono sempre scritti alla fine del file. Lo standard POSIX richiede che i dati scritti siano immediatamente disponibili ad una `read` chiamata dopo che la `write` che li ha scritti è ritornata; ma dati i meccanismi di caching non è detto che tutti i filesystem supportino questa capacità.

Se `count` è zero la funzione restituisce zero senza fare nient'altro. Per i file ordinari il numero di byte scritti è sempre uguale a quello indicato da `count`, a meno di un errore. Negli altri casi si ha lo stesso comportamento di `read`.

Anche per `write` lo standard Unix98 (ed i successivi POSIX.1-2001 e POSIX.1-2008) definiscono un'analoga `pwrite` per scrivere alla posizione indicata senza modificare la posizione corrente nel file, il suo prototipo è:

```
#include <unistd.h>
ssize_t pwrite(int fd, void * buf, size_t count, off_t offset)
```

Scrive a partire da una posizione sul file.

La funzione ritorna il numero di byte letti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà i valori già visti per `write` e `lseek`.

per questa funzione valgono le stesse considerazioni fatte per `pread`, a cui si aggiunge il fatto che su Linux, a differenza di quanto previsto dallo standard POSIX che richiederebbe di ignorarlo, se si è aperto il file con `O_APPEND` i dati saranno comunque scritti in coda al file, ignorando il valore di `offset`.

5.2 Caratteristiche avanzate

In questa sezione approfondiremo alcune delle caratteristiche più sottili della gestione file in un sistema unix-like, esaminando in dettaglio il comportamento delle funzioni base, inoltre tratteremo le funzioni che permettono di eseguire alcune operazioni avanzate con i file (il grosso dell'argomento sarà comunque affrontato nel cap. 10).

5.2.1 La gestione dell'accesso concorrente ai files

In sez. 5.1.1 abbiamo descritto brevemente l'architettura dell'interfaccia con i file da parte di un processo, mostrando in fig. 5.1 le principali strutture usate dal kernel; esamineremo ora in dettaglio le conseguenze che questa architettura ha nei confronti dell'accesso concorrente allo stesso file da parte di processi diversi.

Figura 5.2: Schema dell'accesso allo stesso file da parte di due processi diversi

Il primo caso è quello in cui due processi indipendenti aprono lo stesso file su disco; sulla base di quanto visto in sez. 5.1.1 avremo una situazione come quella illustrata in fig. 5.2: ciascun processo avrà una sua voce nella *file table* referenziata da un diverso file descriptor nella sua *file_struct*. Entrambe le voci nella *file table* faranno però riferimento allo stesso *inode* su disco.

Questo significa che ciascun processo avrà la sua posizione corrente sul file, la sua modalità di accesso e versioni proprie di tutte le proprietà che vengono mantenute nella sua voce della *file table*. Questo ha conseguenze specifiche sugli effetti della possibile azione simultanea sullo stesso file, in particolare occorre tenere presente che:

- ciascun processo può scrivere indipendentemente, dopo ciascuna `write` la posizione corrente sarà cambiata solo nel processo scrivente. Se la scrittura eccede la dimensione corrente del file questo verrà esteso automaticamente con l'aggiornamento del campo `i_size` della struttura *inode*.
- se un file è in modalità `O_APPEND` tutte le volte che viene effettuata una scrittura la posizione corrente viene prima impostata alla dimensione corrente del file letta dalla struttura *inode*. Dopo la scrittura il file viene automaticamente esteso. Questa operazione avviene atomicamente, ogni altro processo che usi `O_APPEND` vedrà la dimensione estesa e continuerà a scrivere in coda al file.
- l'effetto di `lseek` è solo quello di cambiare il campo `f_pos` nella struttura *file* della *file table*, non c'è nessuna operazione sul file su disco. Quando la si usa per porsi alla fine del file la posizione viene impostata leggendo la attuale dimensione corrente dalla struttura *inode*.

Figura 5.3: Schema dell'accesso ai file da parte di un processo figlio

Il secondo caso è quello in cui due file descriptor di due processi diversi puntano alla stessa voce nella *file table*. Questo è ad esempio il caso dei file aperti che vengono ereditati dal processo figlio all'esecuzione di una `fork` (si ricordi quanto detto in sez. 3.1.3). La situazione è illustrata in fig. 5.3; dato che il processo figlio riceve una copia dello spazio di indirizzi del padre, riceverà anche una copia di `file_struct` e della relativa tabella dei file aperti.

Questo significa che il figlio avrà gli stessi file aperti del padre in quanto la sua `file_struct`, pur essendo allocata in maniera indipendente, contiene gli stessi valori di quella del padre e quindi i suoi file descriptor faranno riferimento alla stessa voce nella *file table*, condividendo così la posizione corrente sul file. Questo ha le conseguenze descritte a suo tempo in sez. 3.1.3: in caso di scrittura o lettura da parte di uno dei due processi, la posizione corrente nel file varierà per entrambi, in quanto verrà modificato il campo `f_pos` della struttura `file`, che è la stessa per entrambi. Questo consente una sorta di “sincronizzazione” automatica della posizione sul file fra padre e figlio che occorre tenere presente.

Si noti inoltre che in questo caso anche i flag di stato del file, essendo mantenuti nella struttura `file` della *file table*, vengono condivisi, per cui una modifica degli stessi con `fcntl` (vedi sez. 5.2.5) si applicherebbe a tutti processi che condividono la voce nella *file table*. Ai file però sono associati anche altri flag detti *file descriptor flags*. Questi invece sono mantenuti in `file_struct`, e perciò sono locali per ciascun processo e non vengono modificati dalle azioni degli altri anche in caso di condivisione della stessa voce della *file table*; l'unico usato al momento è quello di *close-on-exec* che indica che il file descriptor deve essere chiuso in una `exec` (vedi sez. 3.1.6).

Si tenga presente dunque che in un sistema unix-like è sempre possibile per più processi accedere in contemporanea allo stesso file e che non esistono, a differenza di altri sistemi operativi, dei meccanismi di blocco o di restrizione dell'accesso impliciti quando più processi vogliono accedere allo stesso file. Questo significa che le operazioni di lettura e scrittura vengono sempre fatte da ogni processo in maniera indipendente, utilizzando una posizione corrente nel file che normalmente, a meno di non trovarsi nella situazione di fig. 5.3, è locale a ciascuno di essi.

Dal punto di vista della lettura dei dati questo comporta la possibilità di poter leggere dati non coerenti in caso di scrittura contemporanea da parte di un altro processo. Dal punto di vista della scrittura invece si potranno avere sovrapposizioni imprevedibili quando due processi scrivono nella stessa sezione di file, dato che ciascuno lo farà in maniera indipendente. Il sistema però fornisce in alcuni casi la possibilità di eseguire alcune operazioni di scrittura in maniera coordinata anche senza utilizzare dei meccanismi di sincronizzazione espliciti come il *file locking*, che esamineremo in sez. 10.1.

Un caso tipico di necessità di accesso condiviso in scrittura è quello in cui vari processi devono scrivere alla fine di un file (ad esempio un file di log). Come accennato in sez. 5.1.3 impostare la posizione alla fine del file con `lseek` e poi scrivere con `write` può condurre ad una *race condition*; infatti può succedere che un secondo processo scriva alla fine del file fra la `lseek` e la `write`. In questo caso, come abbiamo appena visto, il file sarà esteso, ma il primo processo, avrà una posizione corrente che aveva impostato con `lseek` che non corrisponde più alla fine del file, e la sua successiva `write` sovrascriverà i dati del secondo processo.

Il problema deriva dal fatto che usare due *system call* in successione non è mai un'operazione atomica dato che il kernel può interrompere l'esecuzione del processo fra le due. Nel caso specifico il problema è stato risolto introducendo la modalità di scrittura in *append*, attivabile con il flag `O_APPEND`. In questo caso infatti, come abbiamo illustrato in sez. 5.1.2, è il kernel che aggiorna automaticamente la posizione alla fine del file prima di effettuare la scrittura, e poi estende il file. Tutto questo avviene all'interno di una singola *system call*, la `write`, che non essendo interrompibile da un altro processo realizza un'operazione atomica.

5.2.2 La duplicazione dei file descriptor

Abbiamo già visto in sez. 5.2.1 come un processo figlio condivida gli stessi file descriptor del padre; è possibile però ottenere un comportamento analogo all'interno di uno stesso processo con la cosiddetta *duplicazione* di un file descriptor. Per far questo si usa la funzione di sistema `dup`, il cui prototipo è:

<pre>#include <unistd.h> int dup(int oldfd)</pre>	Crea un file descriptor duplicato.
<p>La funzione ritorna il nuovo file descriptor in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EBADF <code>oldfd</code> non è un file aperto.</p> <p>EMFILE si è raggiunto il numero massimo consentito di file descriptor aperti (vedi sez. 6.3.2).</p>	

La funzione ritorna, come `open`, il primo file descriptor libero. Il file descriptor è una copia esatta del precedente ed entrambi possono essere interscambiati nell'uso. Per capire meglio il funzionamento della funzione si può fare riferimento a fig. 5.4. L'effetto della funzione è semplicemente quello di copiare il valore di un certo file descriptor in un altro all'interno della struttura `file_struct`, cosicché anche questo faccia riferimento alla stessa voce nella *file table*. Per questo motivo si dice che il nuovo file descriptor è “*duplicato*”, da cui il nome della funzione.

Figura 5.4: Schema dell'accesso ai file duplicati

Si noti che per quanto illustrato in fig. 5.4 i file descriptor duplicati condivideranno eventuali lock (vedi sez. 10.1), i flag di stato, e la posizione corrente sul file. Se ad esempio si esegue una `lseek` per modificare la posizione su uno dei due file descriptor, essa risulterà modificata anche sull'altro, dato che quello che viene modificato è lo stesso campo nella voce della *file table* a cui entrambi fanno riferimento.

L'unica differenza fra due file descriptor duplicati è che ciascuno avrà un suo *file descriptor flag* indipendente. A questo proposito deve essere tenuto presente che nel caso in cui si usi `dup` per duplicare un file descriptor, se questo ha il flag di *close-on-exec* attivo (vedi sez. 3.1.6 e sez. 5.2.1), questo verrà cancellato nel file descriptor restituito come copia.

L'uso principale di questa funzione è nella shell per la redirezione dei file standard di tab. 5.1 fra l'esecuzione di una `fork` e la successiva `exec`. Diventa così possibile associare un file (o una *pipe*) allo *standard input* o allo *standard output* (vedremo un esempio in sez. 11.1.2, quando tratteremo le *pipe*).

Ci si può chiedere perché non sia in questo caso sufficiente chiudere il file standard che si vuole redirigere e poi aprire direttamente con `open` il file vi si vuole far corrispondere, invece di duplicare un file descriptor che si è già aperto. La risposta sta nel fatto che il file che si vuole redirigere non è detto sia un file regolare, ma potrebbe essere, come accennato, anche una *fifo* o un socket, oppure potrebbe essere un file associato ad un file descriptor che si è ereditato già aperto (ad esempio attraverso una `exec`) da un processo antenato del padre, del quale non si conosce il nome. Operando direttamente con i file descriptor `dup` consente di ignorare le origini del file descriptor che si duplica e funziona in maniera generica indipendentemente dall'oggetto a cui questo fa riferimento.

Per ottenere la redirezione occorre pertanto disporre del file descriptor associato al file che si vuole usare e chiudere il file descriptor che si vuole sostituire, cosicché esso possa esser restituito alla successiva chiamata di `dup` come primo file descriptor disponibile. Dato che

questa è l'operazione più comune, è prevista un'altra funzione di sistema, `dup2`, che permette di specificare esplicitamente qual è il numero di file descriptor che si vuole ottenere come duplicato; il suo prototipo è:

```
#include <unistd.h>
int dup2(int oldfd, int newfd)
```

Duplica un file descriptor su un altro.

La funzione ritorna il nuovo file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF `oldfd` non è un file aperto o `newfd` ha un valore fuori dall'intervallo consentito per i file descriptor.

EBUSY si è rilevata la possibilità di una *race condition*.

EINTR la funzione è stata interrotta da un segnale.

EMFILE si è raggiunto il numero massimo consentito di file descriptor aperti.

La funzione duplica il file descriptor `oldfd` su un altro file descriptor di valore `newfd`. Quando il file descriptor `newfd` sia già aperto, come avviene ad esempio nel caso della duplicazione di uno dei file standard di tab. 5.1, esso sarà prima chiuso e poi duplicato. Se `newfd` è uguale a `oldfd` la funzione non fa nulla e si limita a restituire `newfd`.

L'uso di `dup2` ha vari vantaggi rispetto alla combinazione di `close` e `dup`; anzitutto se `oldfd` è uguale a `newfd` questo verrebbe chiuso e `dup` fallirebbe, ma soprattutto l'operazione è atomica e consente di evitare una *race condition* in cui dopo la chiusura del file si potrebbe avere la ricezione di un segnale il cui gestore (vedi sez. 7.4.5) potrebbe a sua volta aprire un file, per cui alla fine `dup` restituirebbe un file descriptor diverso da quello voluto.

Con Linux inoltre la funzione prevede la possibilità di restituire l'errore `EBUSY`, che non è previsto dallo standard, quando viene rilevata la possibilità di una *race condition* interna in cui si cerca di duplicare un file descriptor che è stato allocato ma per il quale non sono state completate le operazioni di apertura.²¹ In tal caso occorre ritentare l'operazione.

La duplicazione dei file descriptor può essere effettuata anche usando la funzione di controllo dei file `fcntl` (che esamineremo in sez. 5.2.5) con il parametro `F_DUPFD`. L'operazione ha la sintassi `fcntl(oldfd, F_DUPFD, newfd)` e se si usa `0` come valore per `newfd` diventa equivalente a `dup`. La sola differenza fra le due funzioni (a parte la sintassi ed i diversi codici di errore) è che `dup2` chiude il file descriptor `newfd` se questo è già aperto, garantendo che la duplicazione sia effettuata esattamente su di esso, invece `fcntl` restituisce il primo file descriptor libero di valore uguale o maggiore di `newfd`, per cui se `newfd` è aperto la duplicazione avverrà su un altro file descriptor.

Su Linux inoltre è presente una terza funzione di sistema non standard,²² `dup3`, che consente di duplicare un file descriptor reimpostandone i flag, per usarla occorre definire la macro `_GNU_SOURCE` ed il suo prototipo è:

```
#include <unistd.h>
int dup3(int oldfd, int newfd, int flags)
```

Duplica un file descriptor su un altro.

La funzione ritorna il nuovo file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà gli stessi valori di `dup2` più `EINVAL` qualora `flags` contenga un valore non valido o `newfd` sia uguale a `oldfd`.

²¹la condizione è abbastanza peculiare e non attinente al tipo di utilizzo indicato, quanto piuttosto ad un eventuale tentativo di duplicare file descriptor non ancora aperti, la condizione di errore non è prevista dallo standard, ma in condizioni simili FreeBSD risponde con un errore di `EBADF`, mentre OpenBSD elimina la possibilità di una *race condition* al costo di una perdita di prestazioni.

²²la funzione è stata introdotta con il kernel 2.6.27 e resa disponibile con la *glibc* 2.9.

La funzione è identica a `dup2` ma prevede la possibilità di mantenere il flag di *close-on-exec* sul nuovo file descriptor specificando `O_CLOEXEC` in `flags` (che è l'unico flag usabile in questo caso). Inoltre rileva esplicitamente la possibile coincidenza fra `newfd` e `oldfd`, fallendo con un errore di `EINVAL`.

5.2.3 Le funzioni di sincronizzazione dei dati

Come accennato in sez. 5.1.2 tutte le operazioni di scrittura sono in genere bufferizzate dal kernel, che provvede ad effettuarle in maniera asincrona per ottimizzarle, ad esempio accorrendo gli accessi alla stessa zona del disco in un secondo tempo rispetto al momento della esecuzione della *write*.

Per questo motivo quando è necessaria una sincronizzazione immediata dei dati il sistema mette a disposizione delle funzioni che provvedono a forzare lo scarico dei dati dai buffer del kernel. La prima di queste funzioni di sistema è `sync`, il cui prototipo è:²³

```
#include <unistd.h>
void sync(void)
```

Sincronizza il buffer della cache dei file col disco.

La funzione non ritorna nulla e non prevede condizioni di errore.

I vari standard prevedono che la funzione si limiti a far partire le operazioni ritornando immediatamente, con Linux invece, fin dal kernel 1.3.20, la funzione aspetta la conclusione delle operazioni di sincronizzazione. Si tenga presente comunque che questo non dà la garanzia assoluta che i dati siano integri dopo la chiamata, l'hardware dei dischi è in genere dotato di un suo meccanismo interno di bufferizzazione che a sua volta può ritardare ulteriormente la scrittura effettiva.

La funzione viene usata dal comando `sync` quando si vuole forzare esplicitamente lo scarico dei dati su disco, un tempo era invocata da un apposito demone di sistema (in genere chiamato `update`) che eseguiva lo scarico dei dati ad intervalli di tempo fissi. Con le nuove versioni del kernel queste operazioni vengono gestite direttamente dal sistema della memoria virtuale, attraverso opportuni *task* interni al kernel. Nei kernel recenti questo comportamento può essere controllato con l'uso dei vari file `dirty_*` in `/proc/sys/vm/`.²⁴

Si tenga presente che la funzione di sistema `bdflush`, che un tempo veniva usata per controllare lo scaricamento dei dati, è deprecata a partire dal kernel 2.6 e causa semplicemente la stampa di un messaggio nei log del kernel, e non è più presente dalle *glibc* 2.23, pertanto non la prenderemo in esame.

Quando si vogliano scaricare i dati di un singolo file, ad esempio essere sicuri che i dati di un database siano stati registrati su disco, si possono usare le due funzioni di sistema `fsync` e `fdatasync`, i cui prototipi sono:

```
#include <unistd.h>
int fsync(int fd)
```

Sincronizza dati e metadati di un file.

```
int fdatasync(int fd)
```

Sincronizza i dati di un file.

²³questo è il prototipo usato a partire dalla *glibc* 2.2.2 seguendo gli standard, in precedenza la funzione era definita come `int sync(void)` e ritornava sempre 0.

²⁴si consulti la documentazione allegata ai sorgenti del kernel nel file `Documentation/sysctl/vm.txt`, trattandosi di argomenti di natura sistemistica non li prenderemo in esame.

Le funzioni ritornano 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EDQUOT` si è superata una quota disco durante la sincronizzazione.

`EINVAL` `fd` è un file speciale che non supporta la sincronizzazione (talvolta anche `EROFS`).

`EIO` c'è stato un errore di I/O durante la sincronizzazione, che in questo caso può derivare anche da scritture sullo stesso file eseguite su altri file descriptor.

`ENOSPC` si è esaurito lo spazio disco durante la sincronizzazione.

ed inoltre `EBADF` nel suo significato generico.

Entrambe le funzioni forzano la sincronizzazione col disco di tutti i dati del file specificato, ed attendono fino alla conclusione delle operazioni. La prima, `fsync` forza anche la sincronizzazione dei meta-dati del file, che riguardano sia le modifiche alle tabelle di allocazione dei settori, che gli altri dati contenuti nell'*inode* che si leggono con `fstat`, come i tempi del file. Se lo scopo dell'operazione, come avviene spesso per i database, è assicurarsi che i dati raggiungano il disco e siano rileggibili immediatamente in maniera corretta, è sufficiente l'uso di `fdatasync` che evita le scritture non necessarie per avere l'integrità dei dati, come l'aggiornamento dei tempi di ultima modifica ed ultimo accesso.

Si tenga presente che l'uso di queste funzioni non comporta la sincronizzazione della directory che contiene il file e la scrittura della relativa voce su disco, che se necessaria deve essere effettuata esplicitamente con `fsync` sul file descriptor della directory.²⁵

La funzione può restituire anche `ENOSPC` e `EDQUOT` per quei casi in cui l'allocazione dello spazio disco non viene effettuata all'esecuzione di una `write` (come NFS o altri filesystem di rete) per cui l'errore viene rilevato quando la scrittura viene effettivamente eseguita.

L'uso di `sync` può causare, quando ci sono più filesystem montati, problemi di prestazioni dovuti al fatto che effettua la sincronizzazione dei dati su tutti i filesystem, anche quando sarebbe sufficiente eseguirla soltanto su quello dei file su cui si sta lavorando; quando i dati in attesa sono molti questo può causare una alta attività di I/O ed i relativi problemi di prestazioni.

Per questo motivo è stata introdotta una nuova funzione di sistema, `syncfs`,²⁶ che effettua lo scarico dei dati soltanto per il filesystem su cui si sta operando, il suo prototipo è:

```
#include <unistd.h>
int syncfs(int fd)
```

Sincronizza il buffer della cache dei file del singolo filesystem col disco.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` `fd` non è un descrittore valido.

La funzione richiede che si specifichi nell'argomento `fd` un file descriptor su cui si sta operando, e la registrazione immediata dei dati sarà limitata al filesystem su cui il file ad esso corrispondente si trova.

5.2.4 Le *at-functions*: `openat` e le altre

Un problema generico che si pone con l'uso della funzione `open`, così come con le altre funzioni che prendono come argomenti dei *pathname*, è la possibilità, quando si usa un *pathname* che non fa riferimento diretto ad un file posto nella directory di lavoro corrente, che alcuni dei

²⁵in realtà per il filesystem `ext2`, quando lo si monta con l'opzione `sync`, il kernel provvede anche alla sincronizzazione automatica delle voci delle directory.

²⁶la funzione è stata introdotta a partire dal kernel 2.6.39 ed è accessibile solo se è definita la macro `_GNU_SOURCE`, è specifica di Linux e non prevista da nessuno standard.

componenti dello stesso vengano modificati in parallelo alla chiamata a `open`, cosa che lascia aperta la possibilità di una *race condition* in cui c'è spazio per un *symlink attack* (si ricordi quanto visto per `access` in sez. 4.4.3) cambiando una delle directory sovrastanti il file fra un controllo e la successiva apertura.

Inoltre, come già accennato, la directory di lavoro corrente è una proprietà associata al singolo processo; questo significa che quando si lavora con i *thread* questa è la stessa per tutti, per cui se la si cambia all'interno di un *thread* il cambiamento varrà anche per tutti gli altri. Non esiste quindi con le funzioni classiche un modo semplice per far sì che i singoli *thread* possano aprire file usando una propria directory per risolvere i *pathname* relativi.

Per risolvere questi problemi, riprendendo una interfaccia già presente in Solaris, a fianco delle normali funzioni che operano sui file (come `open`, `mkdir`, ecc.) sono state introdotte delle ulteriori funzioni di sistema, chiamate genericamente “*at-functions*” in quanto usualmente contraddistinte dal suffisso `at`, che permettono l'apertura di un file (o le rispettive altre operazioni) usando un *pathname* relativo ad una directory specificata.²⁷ Essendo accomunate dalla stessa interfaccia le tratteremo insieme in questa sezione pur non essendo strettamente attinenti l'I/O su file.

Benché queste funzioni non siano presenti negli standard tradizionali esse sono state adottate da altri sistemi unix-like come Solaris, i vari BSD, fino ad essere incluse in una recente revisione dello standard POSIX.1 (la POSIX.1-2008). Con la *glibc* per l'accesso a queste funzioni è necessario definire la macro `_ATFILE_SOURCE` (comunque attiva di default).

L'uso di queste funzioni richiede una apertura preliminare della directory che si intende usare come base per la risoluzione dei *pathname* relativi (ad esempio usando `open` con il flag `O_PATH` visto in sez. 5.1.2) per ottenere un file descriptor che dovrà essere passato alle stesse. Tutte queste funzioni infatti prevedono la presenza un apposito argomento, in genere il primo che negli esempi seguenti chiameremo sempre `dirfd`, per indicare la directory di partenza.

In questo modo, una volta aperta la directory di partenza, si potranno effettuare controlli ed aperture solo con *pathname* relativi alla stessa, e tutte le *race condition* dovute al possibile cambiamento di uno dei componenti posti al di sopra della stessa cesseranno di esistere. Inoltre, pur restando la directory di lavoro una proprietà comune del processo, si potranno usare queste funzioni quando si lavora con i *thread* per eseguire la risoluzione dei *pathname* relativi ed avere una directory di partenza diversa in ciascuno di essi.

Questo metodo consente inoltre di ottenere aumenti di prestazioni significativi quando si devono eseguire molte operazioni su sezioni dell'albero dei file che prevedono delle gerarchie di sottodirectory molto profonde. Infatti in questo caso basta eseguire la risoluzione del *pathname* di una qualunque directory di partenza una sola volta (nell'apertura iniziale) e non tutte le volte che si deve accedere a ciascun file che essa contiene. Infine poter identificare una directory di partenza tramite il suo file descriptor consente di avere un riferimento stabile alla stessa anche qualora venisse rinominata, e tiene occupato il filesystem dove si trova come per la directory di lavoro di un processo.

La sintassi generica di queste nuove funzioni prevede l'utilizzo come primo argomento del file descriptor della directory da usare come base per la risoluzione dei nomi, mentre gli argomenti successivi restano identici a quelli della corrispondente funzione ordinaria. Come esempio prendiamo in esame la nuova funzione di sistema `openat`, il cui prototipo è:

²⁷L'introduzione è avvenuta su proposta dello sviluppatore principale della *glibc* Ulrich Drepper e le corrispondenti *system call* sono state inserite nel kernel a partire dalla versione 2.6.16, in precedenza era disponibile una emulazione che, sia pure con prestazioni inferiori, funzionava facendo ricorso all'uso del filesystem *proc* con l'apertura del file attraverso il riferimento a *pathname* del tipo di `/proc/self/fd/dirfd/relative_path`.

```
#include <fcntl.h>
int openat(int dirfd, const char *pathname, int flags)
int openat(int dirfd, const char *pathname, int flags, mode_t mode)
                                Apre un file a partire da una directory di lavoro.
```

La funzione ritorna gli stessi valori e gli stessi codici di errore di `open`, ed in più:

EBADF `dirfd` non è un file descriptor valido.

ENOTDIR `pathname` è un *pathname* relativo, ma `dirfd` fa riferimento ad un file.

Il comportamento di `openat` è del tutto analogo a quello di `open`, con la sola eccezione del fatto che se per l'argomento `pathname` si utilizza un *pathname* relativo questo sarà risolto rispetto alla directory indicata da `dirfd`; qualora invece si usi un *pathname* assoluto `dirfd` verrà semplicemente ignorato. Infine se per `dirfd` si usa il valore speciale `AT_FDCWD` la risoluzione sarà effettuata rispetto alla directory di lavoro corrente del processo. Questa, come le altre costanti `AT_*`, è definita in `fcntl.h`, per cui per usarla occorrerà includere comunque questo file, anche per le funzioni che non sono definite in esso.

Così come il comportamento, anche i valori di ritorno e le condizioni di errore delle nuove funzioni sono gli stessi delle funzioni classiche, agli errori si aggiungono però quelli dovuti a valori errati per `dirfd`; in particolare si avrà un errore di `EBADF` se esso non è un file descriptor valido, ed un errore di `ENOTDIR` se esso non fa riferimento ad una directory, tranne il caso in cui si sia specificato un *pathname* assoluto, nel qual caso, come detto, il valore di `dirfd` sarà completamente ignorato.

Funzione	Flags	Corrispondente
<code>execveat</code>	●	<code>execve</code>
<code>faccessat</code>	●	<code>access</code>
<code>fchmodat</code>	●	<code>chmod</code>
<code>fchownat</code>	●	<code>chown, lchown</code>
<code>fstatat</code>	●	<code>stat, lstat</code>
<code>futimesat</code>	—	obsoleta
<code>linkat</code>	●	<code>link</code>
<code>mkdirat</code>	—	<code>mkdir</code>
<code>mkfifoat</code>	—	<code>mkfifo</code>
<code>mknodat</code>	—	<code>mknod</code>
<code>openat</code>	—	<code>open</code>
<code>readlinkat</code>	—	<code>readlink</code>
<code>renameat</code>	—	<code>rename</code>
<code>renameat2</code> ²⁸	—	<code>rename</code>
<code>scandirat</code>	—	<code>scandir</code>
<code>statx</code>	●	<code>stat</code>
<code>symlinkat</code>	—	<code>symlink</code>
<code>unlinkat</code>	●	<code>unlink, rmdir</code>
<code>utimensat</code>	●	<code>utimes, lutimes</code>

Tabella 5.6: Corrispondenze fra le nuove funzioni “at” e le corrispettive funzioni classiche.

In tab. 5.6 si sono riportate le funzioni introdotte con questa nuova interfaccia, con a fianco la corrispondente funzione classica. Tutte seguono la convenzione appena vista per `openat`, in cui agli argomenti della funzione classica viene anteposto l'argomento `dirfd`. Per alcune, indicate dal contenuto della omonima colonna di tab. 5.6, oltre al nuovo argomento iniziale, è prevista anche l'aggiunta di un argomento finale, `flags`, che è stato introdotto per fornire un meccanismo con cui modificarne il comportamento.

²⁸anche se la funzione ha un argomento `flags` questo attiene a funzionalità specifiche della stessa e non all'uso generico fatto nelle altre *at-functions*, pertanto lo si è indicato come assente.

Per tutte quelle che non hanno un argomento aggiuntivo il comportamento è identico alla corrispondente funzione ordinaria, pertanto non le tratteremo esplicitamente, vale per loro quanto detto con `openat` per l'uso del nuovo argomento `dirfd`. Tratteremo invece esplicitamente tutte quelle per cui l'argomento è presente, in quanto il loro comportamento viene modificato a seconda del valore assegnato a `flags`; questo deve essere passato come maschera binaria con una opportuna combinazione delle costanti elencate in tab. 5.7, in quanto sono possibili diversi valori a seconda della funzione usata.

Costante	Significato
AT_EMPTY_PATH	Usato per operare direttamente (specificando una stringa vuota per il <code>pathname</code>) sul file descriptor <code>dirfd</code> che in questo caso può essere un file qualunque.
AT_SYMLINK_NOFOLLOW	Se impostato la funzione non esegue la dereferenziazione dei collegamenti simbolici.
AT_EACCESS	Usato solo da <code>faccessat</code> , richiede che il controllo dei permessi sia fatto usando l' <code>UID</code> effettivo invece di quello reale.
AT_NO_AUTOMOUNT	Usato solo da <code>fstatat</code> e <code>statx</code> , evita il montaggio automatico qualora <code>pathname</code> faccia riferimento ad una directory marcata per l' <i>automount</i> ²⁹ (dal kernel 2.6.38).
AT_REMOVEDIR	Usato solo da <code>unlinkat</code> , richiede che la funzione si comporti come <code>rmdir</code> invece che come <code>unlink</code> .
AT_SYMLINK_FOLLOW	Usato solo da <code>linkat</code> , se impostato la funzione esegue la dereferenziazione dei collegamenti simbolici.

Tabella 5.7: Le costanti utilizzate per i bit dell'argomento aggiuntivo `flags` delle *at-functions*, definite in `fcntl.h`.

Si tenga presente che non tutte le funzioni che prevedono l'argomento aggiuntivo sono *system call*, ad esempio `faccessat` e `fchmodat` sono realizzate con dei *wrapper* nella *glibc* per aderenza allo standard POSIX.1-2008, dato che la *system call* sottostante non prevede l'argomento `flags`.

In tab. 5.7 si sono elencati i valori utilizzabili per i flag (tranne quelli specifici di `statx` su cui torneremo più avanti), mantenendo nella prima parte quelli comuni usati da più funzioni. Il primo di questi è `AT_SYMLINK_NOFOLLOW`, che viene usato da tutte le funzioni tranne `linkat` e `unlinkat`, e che consente di scegliere, quando si sta operando su un collegamento simbolico, se far agire la funzione direttamente sullo stesso o sul file da esso referenziato. Si tenga presente però che per `fchmodat` questo, che è l'unico flag consentito e previsto dallo standard, non è attualmente implementato (anche perché non avrebbe molto senso cambiare i permessi di un link simbolico) e pertanto l'uso della funzione è analogo a quello delle altre funzioni che non hanno l'argomento `flags` (e non la tratteremo esplicitamente).

L'altro flag comune è `AT_EMPTY_PATH`, utilizzabile a partire dal kernel 2.6.39, che consente di usare per `dirfd` un file descriptor associato ad un file qualunque e non necessariamente ad una directory; in particolare si può usare un file descriptor ottenuto aprendo un file con il flag `O_PATH` (vedi quanto illustrato a pag. 205). Quando si usa questo flag `pathname` deve essere vuoto, da cui il nome della costante, ed in tal caso la funzione agirà direttamente sul file associato al file descriptor `dirfd`.

Una prima funzione di sistema che utilizza l'argomento `flag` è `fchownat`, che può essere usata per sostituire sia `chown` che `lchown`; il suo prototipo è:

²⁹L'*automount* è una funzionalità fornita dal kernel che consente di montare automaticamente una directory quando si accede ad un *pathname* al di sotto di essa, per i dettagli, di natura prevalentemente sistemistica, si può consultare sez. 5.1.6 di [?].

```
#include <fcntl.h>
#include <unistd.h>
int fchownat(int dirfd, const char *pathname, uid_t owner, gid_t group, int flags)
                                                    Modifica il proprietario di un file.
```

La funzione ritorna gli stessi valori e gli stessi codici di errore di `chown`, ed in più:

EBADF `dirfd` non è un file descriptor valido.

EINVAL `flags` non ha un valore valido.

ENOTDIR `pathname` è un *pathname* relativo, ma `dirfd` fa riferimento ad un file.

In questo caso, oltre a quanto già detto per `openat` riguardo all'uso di `dirfd`, se si è impostato `AT_SYMLINK_NOFOLLOW` in `flags`, si indica alla funzione di non eseguire la dereferenziazione di un eventuale collegamento simbolico, facendo comportare `fchownat` come `lchown` invece che come `chown`. La funzione supporta anche l'uso di `AT_EMPTY_PATH`, con il significato illustrato in precedenza e non ha flag specifici.

Una seconda funzione di sistema che utilizza l'argomento `flags`, in questo caso anche per modificare il suo comportamento, è `faccessat`, ed il suo prototipo è:

```
#include <fcntl.h>
#include <unistd.h>
int faccessat(int dirfd, const char *path, int mode, int flags)
                                                    Controlla i permessi di accesso.
```

La funzione ritorna gli stessi valori e gli stessi codici di errore di `access`, ed in più:

EBADF `dirfd` non è un file descriptor valido.

EINVAL `flags` non ha un valore valido.

ENOTDIR `pathname` è un *pathname* relativo, ma `dirfd` fa riferimento ad un file.

La funzione esegue il controllo di accesso ad un file, e `flags` consente di modificarne il comportamento rispetto a quello ordinario di `access` (cui è analoga e con cui condivide i problemi di sicurezza visti in sez. 4.3.1) usando il valore `AT_EACCESS` per indicare alla funzione di eseguire il controllo dei permessi con l'*UID effettivo* invece di quello *reale*. L'unico altro valore consentito è `AT_SYMLINK_NOFOLLOW`, con il significato già spiegato.

Un utilizzo specifico dell'argomento `flags` viene fatto anche dalla funzione di sistema `unlinkat`, in questo caso l'argomento viene utilizzato perché tramite esso si può indicare alla funzione di comportarsi sia come analogo di `unlink` che di `rmdir`; il suo prototipo è:

```
#include <fcntl.h>
#include <unistd.h>
int unlinkat(int dirfd, const char *pathname, int flags)
                                                    Rimuove una voce da una directory.
```

La funzione ritorna gli stessi valori e gli stessi codici di errore di `unlink` o di `rmdir` a seconda del valore di `flags`, ed in più:

EBADF `dirfd` non è un file descriptor valido.

EINVAL `flags` non ha un valore valido.

ENOTDIR `pathname` è un *pathname* relativo, ma `dirfd` fa riferimento ad un file.

Di default il comportamento di `unlinkat` è equivalente a quello che avrebbe `unlink` applicata a `pathname`, fallendo in tutti i casi in cui questo è una directory, se però si imposta `flags` al valore di `AT_REMOVEDIR`, essa si comporterà come `rmdir`, in tal caso `pathname` deve essere una directory, che sarà rimossa qualora risulti vuota. Non essendo in questo caso prevista la possibilità di usare altri valori (la funzione non segue comunque i collegamenti simbolici e `AT_EMPTY_PATH` non è supportato) anche se `flags` è una maschera binaria, essendo `AT_REMOVEDIR` l'unico flag disponibile per questa funzione, lo si può assegnare direttamente.

Un'altra funzione di sistema che usa l'argomento `flags` è `utimensat`, che però non è una corrispondente esatta delle funzioni classiche `utimes` e `lutimes`, in quanto ha una maggiore precisione nella indicazione dei tempi dei file, per i quali, come per `futimens`, si devono usare strutture `timespec` che consentono una precisione fino al nanosecondo; la funzione è stata introdotta con il kernel 2.6.22,³⁰ ed il suo prototipo è:

```
#include <fcntl.h>
#include <sys/stat.h>
int utimensat(int dirfd, const char *pathname, const struct timespec times[2],
              int flags)
```

Cambia i tempi di un file.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà i valori di `utimes`, `lutimes` e `futimens` con lo stesso significato ed inoltre:

EBADF `dirfd` non è `AT_FDCWD` o un file descriptor valido.
EFAULT `dirfd` è `AT_FDCWD` ma `pathname` è `NULL` o non è un puntatore valido.
EINVAL si usato un valore non valido per `flags`, oppure `pathname` è `NULL`, `dirfd` non è `AT_FDCWD` e `flags` contiene `AT_SYMLINK_NOFOLLOW`.
ESRCH non c'è il permesso di attraversamento per una delle componenti di `pathname`.

La funzione imposta i tempi dei file utilizzando i valori passati nel vettore di strutture `timespec` ed ha in questo lo stesso comportamento di `futimens`, vista in sez. 4.3.4, ma al contrario di questa può essere applicata anche direttamente ad un file come `utimes`; l'unico valore consentito per `flags` è `AT_SYMLINK_NOFOLLOW` che indica alla funzione di non dereferenziare i collegamenti simbolici, cosa che le permette di riprodurre anche le funzionalità di `lutimes` (con una precisione dei tempi maggiore).

Su Linux solo `utimensat` è una *system call* mentre `futimens` è una funzione di libreria, infatti `utimensat` ha un comportamento speciale se `pathname` è `NULL`, in tal caso `dirfd` viene considerato un file descriptor ordinario e il cambiamento del tempo viene applicato al file sottostante, qualunque esso sia. Viene cioè sempre usato il comportamento che per altre funzioni deve essere attivato con `AT_EMPTY_PATH` (che non è previsto per questa funzione) per cui `futimens(fd, times)` è del tutto equivalente a `utimensat(fd, NULL, times, 0)`. Si tenga presente che nella *glibc* questo comportamento è disabilitato, e la funzione, seguendo lo standard POSIX, ritorna un errore di `EINVAL` se invocata in questo modo.

Come corrispondente di `stat`, `fstat` e `lstat` si può utilizzare invece la funzione di sistema `fstatat`, il cui prototipo è:

```
#include <fcntl.h>
#include <sys/stat.h>
int fstatat(int dirfd, const char *pathname, struct stat *statbuf, int flags)
```

Legge le informazioni di un file.

La funzione ritorna gli stessi valori e gli stessi codici di errore di `stat`, `fstat`, o `lstat` a seconda del valore di `flags`, ed in più:

EBADF `dirfd` non è un file descriptor valido.
EINVAL `flags` non ha un valore valido.
ENOTDIR `pathname` è un *pathname* relativo, ma `dirfd` fa riferimento ad un file.

La funzione ha lo stesso comportamento delle sue equivalenti classiche, l'uso di `flags` consente di farla comportare come `lstat` se si usa `AT_SYMLINK_NOFOLLOW`, o come `fstat` se si usa con `AT_EMPTY_PATH` e si passa il file descriptor in `dirfd`. Viene però supportato l'ulteriore

³⁰in precedenza, a partire dal kernel 2.6.16, era stata introdotta una *system call* `futimesat` seguendo una bozza della revisione dello standard poi modificata; questa funzione, sostituita da `utimensat`, è stata dichiarata obsoleta, non è supportata da nessuno standard e non deve essere più utilizzata: pertanto non ne parleremo.

valore `AT_NO_AUTOMOUNT` che qualora `pathname` faccia riferimento ad una directory marcata per l'*automount* ne evita il montaggio automatico.

Ancora diverso è il caso di `linkat` anche se in questo caso l'utilizzo continua ad essere attinente al comportamento con i collegamenti simbolici, il suo prototipo è:

```
#include <fcntl.h>
int linkat(int olddirfd, const char *oldpath, int newdirfd,
           const char *newpath, int flags)
           Crea un nuovo collegamento diretto (hard link).
```

La funzione ritorna gli stessi valori e gli stessi codici di errore di `link`, ed in più:

<code>EBADF</code>	<code>olddirfd</code> o <code>newdirfd</code> non sono un file descriptor valido.
<code>EINVAL</code>	<code>flags</code> non ha un valore valido.
<code>ENOENT</code>	<code>oldpath</code> o <code>newpath</code> è un <i>pathname</i> relativo, ma la corrispondente directory di partenza (<code>olddirfd</code> o <code>newdirfd</code>) è stata cancellata, oppure si è cercato di creare un <i>link</i> da un file descriptor aperto con <code>O_TMPFILE</code> e <code>O_EXCL</code> , oppure si è usato <code>AT_EMPTY_PATH</code> senza privilegi amministrativi.
<code>ENOTDIR</code>	<code>oldpath</code> e <code>newpath</code> sono <i>pathname</i> relativi, ma <code>olddirfd</code> o <code>newdirfd</code> fa riferimento ad un file.
<code>EPERM</code>	si è usato <code>AT_EMPTY_PATH</code> con <code>oldpath</code> vuoto e <code>olddirfd</code> che fa riferimento ad una directory.

Anche in questo caso la funzione svolge lo stesso compito della corrispondente classica `link`, ma dovendo specificare due *pathname* (sorgente e destinazione) aggiunge a ciascuno di essi un argomento (rispettivamente `olddirfd` e `newdirfd`) per poter indicare entrambi come relativi a due directory aperte in precedenza.

In questo caso, dato che su Linux il comportamento di `link` è quello di non seguire mai i collegamenti simbolici, `AT_SYMLINK_NOFOLLOW` non viene utilizzato. A partire dal kernel 2.6.18 è stato aggiunto a questa funzione la possibilità di usare il valore `AT_SYMLINK_FOLLOW` per l'argomento `flags`,³¹ che richiede di dereferenziare un eventuale collegamento simbolico creando un *hard link* al file puntato da quest'ultimo.

Inoltre a partire dal kernel 3.11 si può usare `AT_EMPTY_PATH` con lo stesso significato già visto in precedenza applicato ad `olddirfd`, si può cioè creare un nuovo *hard link* al file associato al file descriptor `olddirfd`, passando un valore nullo per `oldpath`. Questa operazione però è privilegiata e richiede i privilegi di amministratore (la *capability* `CAP_DAC_READ_SEARCH`), infatti in questo modo la funzione si comporta come una ipotetica `flink`, una *system call* di cui è stato spesso chiesta la creazione, che permetterebbe di associare direttamente un nome ad un file descriptor, ma che non è mai stata realizzata per problemi di sicurezza.

Il problema infatti è che le verifiche di accesso sono fatte quando il file viene aperto e non attengono solo ai permessi del file stesso, ma anche a quelli delle directory del suo *pathname*; se una volta aperto venisse collegato in un'altra directory eventuali restrizioni imposte sulle directory del suo *pathname* andrebbero perse. Inoltre sarebbe possibile accedere al file sottostante anche in scrittura per un file descriptor che è stato fornito come aperto in sola lettura, o con accesso libero per un file descriptor fornito aperto in *append*. Infine la funzione consentirebbe rendere accessibile all'interno di un *choot* (vedi sez. 4.5.4) un qualunque file sia stato aperto fuori dallo stesso prima di entrarvi.

Per questo motivo l'uso di `AT_EMPTY_PATH` richiede comunque privilegi amministrativi, anche se, quando è disponibile il filesystem `/proc`, è possibile usare `linkat` per creare un file

³¹nei kernel precedenti, dall'introduzione nel 2.6.16, l'argomento `flags` era presente, ma senza alcun valore valido, e doveva essere passato sempre con valore nullo.

da un qualunque file descriptor un processo abbia aperto, usandola con un codice analogo al seguente:³²

```
linkat(AT_FDCWD, "/proc/self/fd/3", dirfd, file, AT_SYMLINK_FOLLOW);
```

e questa modalità è anche quella con cui è possibile assegnare in un secondo tempo il nome ad un file anonimo creato usando `open` con `O_TMPFILE`; ma si deve tenere presente che per questi file la funzione ha un comportamento particolare.

In generale infatti quando il file sorgente di `linkat` ha un numero di collegamenti nulli (cosa che avviene ad esempio quando si apre un file temporaneo e lo si cancella subito dopo oppure quando viene cancellato un file aperto in precedenza) la funzione non consente di ricollegarlo ad un altro file riassegnandogli un nuovo nome e fallisce sempre con un errore di `ENOENT` qualunque siano i permessi del processo, e che si usi questo approccio o `AT_EMPTY_PATH`. Ma questo non avviene se il file descriptor è stato ottenuto con `O_TMPFILE`, in tal caso la funzione ha successo, a meno che non si sia usato nell'apertura anche `O_EXCL` per impedire questo comportamento, e continuare ad ottenere `ENOENT`.

In fig. 5.5 si è riportato il codice della funzione `InitFile`, che consente di creare in maniera sicura il contenuto iniziale di un file utilizzando `O_TMPFILE` e `linkat`, come accennato a pag. 203. La funzione richiede di indicare il file da creare usando la sintassi delle *at-functions*, specificando la directory in cui crearlo con il corrispondente file descriptor passato nell'argomento `dirfd` ed il pathname relativo ed essa passato l'argomento `file`; il contenuto iniziale del file deve essere fornito nel buffer `buf` di lunghezza `size`.

```

1 ssize_t InitFile(int dirfd, const char *file, const char *buf, size_t size)
2 {
3     int fd, written, res;
4     char path[PATH_MAX];
5
6     fd = openat(dirfd, ".", O_TMPFILE|O_RDWR, S_IRUSR|S_IWUSR);
7     if (fd < 0) {
8         perror("Cannot_get_temporary_filedescriptor");
9         return(fd);
10    }
11    written = FullWrite(fd, buf, size);
12    if (written < 0) {
13        perror("error_writing_on_tmp_file");
14        return(res);
15    }
16    snprintf(path, PATH_MAX, "/proc/self/fd/%d", fd);
17    res = linkat(AT_FDCWD, path, dirfd, file, AT_SYMLINK_FOLLOW);
18    if (res < 0) {
19        perror("error_linking_the_file");
20        return(res);
21    } else {
22        return written;
23    }
24 }

```

Figura 5.5: Esempio di codice per creare in maniera sicura il contenuto iniziale di un file.

La funzione come primo passo (6-10) ottiene un file descriptor accessibile in lettura/scrittura invocando `openat` con il flag `O_TMPFILE` per ottenere un file anonimo, facendo riferimento

³²non esiste al momento, se si sta usando il filesystem *proc*, una modalità per evitare i rischi illustrati in precedenza.

a quella che sarà la directory di destinazione in cui poi verrà collegato lo stesso passata dal chiamante in `dirfd`, usando “.” come *pathname* relativo. Si noti come nella chiamata si impostino anche (per semplicità si è usato un valore fisso) i valori iniziali dei permessi del file (lettura e scrittura solo per il proprietario), e come dopo la chiamata si controlli la presenza di un eventuale errore, ritornandolo con un messaggio qualora avvenga.

Il secondo passo (11-15) è quello di chiamare la funzione `FullWrite` (che tratteremo in dettaglio in sez. 15.3.1) per eseguire la scrittura del contenuto del buffer `buf` sul file anonimo ottenuto con `openat`; in sostanza la funzione scrive tutto il contenuto del buffer, iterando le scritture qualora non sia possibile eseguire tutto con una singola `write`, cosa che comunque per i file su disco in genere non avviene mai.

Una volta completata con successo la scrittura l'ultimo passo (17-23) è collegare il file anonimo con `linkat`, per questo però occorre utilizzare il *pathname* ad esso associato sotto `/proc`, che viene ottenuto (16) con una `snprintf` (vedi sez. 5.3.6) usando file descriptor restituito da `openat`. Con questo *pathname* si può procedere (17) a chiamare `linkat` per eseguire il collegamento, in cui occorre usare il flag `AT_SYMLINK_NOFOLLOW` come nell'esempio precedente.

Altre due funzioni che utilizzano due *pathname* (e due file descriptor) sono `renameat` e `renameat2`, corrispondenti alla classica `rename`; i rispettivi prototipi sono:

```
#include <fcntl.h>
int renameat(int olddirfd, const char *oldpath, int newdirfd, const char *newpath)
int renameat2(int olddirfd, const char *oldpath, int newdirfd,
              const char *newpath, int flags)
```

Rinomina o sposta un file o una directory.

Le funzioni ritornano gli stessi valori e gli stessi codici di errore di `rename`, ed in più per entrambe:

EBADF `olddirfd` o `newdirfd` non sono un file descriptor valido.

ENOTDIR `oldpath` e `newpath` sono *pathname* relativi, ma i corrispondenti `olddirfd` o `newdirfd` fan riferimento ad un file e non a una directory.

e per `renameat2` anche:

EEXIST si è richiesto `RENAME_NOREPLACE` ma `newpath` esiste già.

EINVAL Si è usato un flag non valido in `flags`, o si sono usati insieme a `RENAME_EXCHANGE` o `RENAME_NOREPLACE` o `RENAME_WHITEOUT`, o non c'è il supporto nel filesystem per una delle operazioni richieste in `flags`.

ENOENT si è richiesto `RENAME_EXCHANGE` e `newpath` non esiste.

EPERM si è richiesto `RENAME_WHITEOUT` ma il chiamante non ha i privilegi di amministratore.

In realtà la corrispondente di `rename`, prevista dallo standard POSIX.1-2008 e disponibile dal kernel 2.6.16 come le altre *at-functions*, sarebbe soltanto `renameat`, su Linux però, a partire dal kernel dal 3.15, questa è stata realizzata in termini della nuova funzione di sistema `renameat2` che prevede l'uso dell'argomento aggiuntivo `flags`; in questo caso `renameat` è totalmente equivalente all'utilizzo di `renamat2` con un valore nullo per `flags`.

L'uso di `renameat` è identico a quello di `rename`, con la sintassi delle *at-functions* applicabile ad entrambi i *pathname* passati come argomenti alla funzione. Con `renameat2` l'introduzione dell'argomento `flags` (i cui valori possibili sono riportati in tab. 5.8) ha permesso di aggiungere alcune funzionalità specifiche di Linux non previste al momento da nessuno standard (la funzione è disponibile nelle *glibc* a partire dalla versione 2.28).

L'uso dell'argomento `flags` in questo caso non attiene alle funzionalità relative alla *at-functions*, ma consente di estendere le funzionalità di `rename`. In particolare `renameat2` consente di eseguire uno scambio di nomi in maniera atomica usando il flag `RENAME_EXCHANGE`; se specificato la funzione rinomina in un colpo solo `oldpath` in `newpath` e `newpath` in `oldpath`. Usando questo flag, entrambi i *pathname* passati come argomenti devono esistere, e non è pos-

Costante	Significato
RENAME_EXCHANGE	richiede uno scambio di nomi fra <i>oldpath</i> e <i>newpath</i> , non è usabile con RENAME_NOREPLACE.
RENAME_NOREPLACE	non sovrascrive <i>newpath</i> se questo esiste dando un errore.
RENAME_WHITEOUT	crea un oggetto di <i>whiteout</i> contestualmente al cambio di nome (disponibile a partire dal kernel 3.18).

Tabella 5.8: I valori specifici dei bit dell'argomento *flags* per l'uso con *renameat2*.

sibile usare RENAME_NOREPLACE, non ci sono infine restrizioni sul tipo di file (regolare, directory, link simbolici, dispositivo) di cui si scambia il nome.

Il flag RENAME_NOREPLACE consente di richiedere la generazione di un errore nei casi in cui *rename* avrebbe causato una sovrascrittura della destinazione, rendendo possibile evitare la stessa in maniera atomica; un controllo preventivo dell'esistenza del file infatti avrebbe aperto alla possibilità di una *race condition* fra il momento del controllo e quella del cambio di nome.

Infine il flag RENAME_WHITEOUT, introdotto con il kernel 3.18, richiede un approfondimento specifico, in quanto attiene all'uso della funzione con dei filesystem di tipo *overlay/union*, dato che il flag ha senso solo quando applicato a file che stanno su questo tipo di filesystem. Un *overlay* o *union filesystem* è un filesystem speciale strutturato in livelli, in cui si rende scrivibile un filesystem accessibile in sola lettura, *sovrapponendogli* un filesystem scrivibile su cui vanno tutte le modifiche. Un tale tipo di filesystem serve ad esempio a rendere scrivibili i dati processati quando si fa partire una distribuzione *Live* basata su CD o DVD, ad esempio usando una chiavetta o uno spazio disco aggiuntivo.

In questo caso quando si rinomina un file che sta nello strato in sola lettura questo viene copiato a destinazione sulla parte accessibile in scrittura, ma l'originale non può essere cancellato; per far sì che esso non appaia più è possibile creare un oggetto speciale del filesystem, chiamato *whiteout*, che serve a renderlo non più visibile. La funzione consente di creare questo oggetto, che in un filesystem ordinario verrebbe visto come un file di dispositivo con *major minor* e *minor number* nulli, in maniera atomica quando si rinomina un file. Dato che l'uso di RENAME_WHITEOUT comporta in sostanza la creazione di un file di dispositivo, l'operazione è privilegiata (occorre la *capability* CAP_MKNOD), inoltre occorre anche il supporto nel filesystem usato come supporto per la scrittura. Infine l'operazione non è compatibile con RENAME_EXCHANGE.

Benché non rientri nelle *at-functions* previste nello standard POSIX.1-2008, tratteremo qui anche la funzione di sistema *statx*, introdotta con il kernel 4.11 e disponibile dalle versione 2.28 della *glibc*, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int statx(int dirfd, const char *pathname, int flags,
          unsigned int mask, struct statx *statxbuf)
```

Legge le informazioni di un file.

La funzione ritorna gli stessi valori e gli stessi codici di errore di *stat*, *fstat*, o *lstat* a seconda del valore di *flags*, ed in più:

EBADF *dirfd* non è un file descriptor valido.

EINVAL *flags* non ha un valore valido o *mask* ha un valore riservato.

ENOTDIR *pathname* è un *pathname* relativo, ma *dirfd* fa riferimento ad un file.

La funzione è una estensione specifica di Linux consente di leggere le informazioni di un file; ha la stessa sintassi di *fstatat* utilizzando con lo stesso significato gli argomenti *dirfd* e

pathname ed i valori `AT_EMPTY_PATH`, `AT_NO_AUTOMOUNT` e `AT_SYMLINK_NOFOLLOW` per `flags`. Si può pertanto indicare il file di cui si vogliono ottenere i dati con un *pathname* assoluto, con un *pathname* relativo (sia alla directory corrente che a quella indicata da `dirfd`) o con un file descriptor ad esso associato.

La funzione però consente di ottenere informazioni più dettagliate rispetto a quelle fornite dalle funzioni tradizionali come `stat` e `fstatat`, ed è in grado di controllare le modalità con cui le ottiene nel caso un file sia posto su un filesystem remoto. Per questo, oltre ai tre valori precedenti, l'argomento `flags` consente anche gli ulteriori valori elencati in tab. 5.9, con il significato ivi illustrato.

Costante	Significato
<code>AT_STATX_SYNC_AS_STAT</code>	si comporta esattamente come <code>stat</code> , in questo caso (il default se non viene indicato niente) il risultato dipende dal tipo di filesystem.
<code>AT_STATX_FORCE_SYNC</code>	richiede che i valori degli attributi richiesti siano, in caso di un filesystem di rete, siano sincronizzati con il server remoto, questo può forzare una scrittura dei dati (in particolare i tempi del file) verso lo stesso.
<code>AT_STATX_DONT_SYNC</code>	chiede di non sincronizzare nessun dato, ritornando quanto presente nella cache, questo significa che i dati potrebbero essere non coerenti ed aggiornati, ma si evita, in caso di filesystem di rete, la necessità di contattare il server remoto.

Tabella 5.9: Valori specifici di `statx` per l'argomento `flags`.

La funzione restituisce le informazioni relative al file richiesto nella struttura `statx` puntata dall'argomento `statxbuf`. Inoltre data la quantità di informazioni che possono essere richieste, la funzione consente, con l'argomento `mask` di selezionare quelle volute, questa deve essere assegnata ad una maschera binaria dei valori illustrati in tab. 5.10.

Costante	Significato
<code>STATX_TYPE</code>	Tipo del file (<code>stx_mode</code> & <code>S_IFMT</code>).
<code>STATX_MODE</code>	Permessi del file (<code>stx_mode</code> & <code>~S_IFMT</code>).
<code>STATX_NLINK</code>	Numero di collegamenti (<i>hard link</i> , <code>stx_nlink</code>).
<code>STATX_UID</code>	Proprietario del file (per <i>UID</i> , <code>stx_uid</code>).
<code>STATX_GID</code>	Gruppo proprietario del file (per <i>GID</i> , <code>stx_gid</code>).
<code>STATX_ATIME</code>	Tempo di ultimo accesso (<code>stx_atime</code>).
<code>STATX_MTIME</code>	Tempo di ultima modifica (<code>stx_mtime</code>).
<code>STATX_CTIME</code>	Tempo di ultimo cambiamento (<code>stx_ctime</code>).
<code>STATX_INO</code>	Numero di <i>inode</i> (<code>stx_ino</code>).
<code>STATX_SIZE</code>	Dimensione del file (<code>stx_size</code>).
<code>STATX_BLOCKS</code>	Numero di blocchi del file (<code>stx_blocks</code>).
<code>STATX_BASIC_STATS</code>	Tutte le informazioni precedenti.
<code>STATX_BTIME</code>	Tempo di creazione (<code>stx_btime</code>).
<code>STATX_ALL</code>	Tutte le informazioni.

Tabella 5.10: Le costanti per i valori dell'argomento `mask` di `statx`.

Si tenga presente che il kernel non richiede che `mask` contenga solo i flag di tab. 5.10, valori ulteriori in genere vengono ignorati ma non si può comunque indicare un valore qualunque in quanto alcuni bit sono riservati per future estensioni.³³ Inoltre non è detto che tutte le informazioni richieste con `mask` siano disponibili, per questo il kernel restituisce in un opportuno campo della struttura `statx`, `stx_mask`, quali sono i dati effettivamente restituiti, che possono in alcuni casi essere anche di più di quelli richiesti (se l'informazione aggiuntiva è

³³in particolare il bit `STATX__RESERVED` che se usato causa il fallimento della funzione con un errore di `EINVAL`.

ottenuta senza costi ulteriori) per cui è normale che questo valore possa essere diverso da quanto richiesto.

```

struct statx {
    __u32 stx_mask;           /* Mask of bits indicating filled fields */
    __u32 stx_blksize;        /* Preferred block size for filesystem I/O */
    __u64 stx_attributes;     /* Extra file attribute indicators */
    __u32 stx_nlink;          /* Number of hard links */
    __u32 stx_uid;            /* User ID of owner */
    __u32 stx_gid;            /* Group ID of owner */
    __u16 stx_mode;           /* File type and mode */
    __u16 __spare0[1];
    __u64 stx_ino;            /* Inode number */
    __u64 stx_size;           /* File size in bytes */
    __u64 stx_blocks;         /* Number of 512-byte blocks allocated */
    __u64 stx_attributes_mask; /* Mask to show what's supported in stx_attributes */
    /* The following fields are file timestamps */
    struct statx_timestamp stx_atime; /* Last access time */
    struct statx_timestamp stx_btime; /* File creation time */
    struct statx_timestamp stx_ctime; /* Last status change time */
    struct statx_timestamp stx_mtime; /* Last data modification time */
    /* If this file represents a device, then the next two
       fields contain the ID of the device */
    __u32 stx_rdev_major;     /* Major ID */
    __u32 stx_rdev_minor;     /* Minor ID */
    /* The next two fields contain the ID of the device
       containing the filesystem where the file resides */
    __u32 stx_dev_major;      /* Major ID */
    __u32 stx_dev_minor;      /* Minor ID */
    __u64 __spare2[14];       /* Spare space for future expansion */
};

```

Figura 5.6: La struttura `statx` per la lettura delle informazioni dei file.

Si è riportata in fig. 5.6 la definizione della struttura `statx` come presente in `sys/stat.h`; i campi `stx_mode`, `stx_nlink`, `stx_uid`, `stx_gid`, `stx_ino`, `stx_size`, `stx_blksize`, `stx_blocks` sono identici agli analoghi (con prefisso `st_`) dell'ordinaria struttura `stat` illustrata in fig. 4.13 e vale per essi quanto già detto in sez. 4.3.1 e seguenti.

```

struct statx_timestamp {
    __s64 tv_sec;           /* Seconds since the Epoch (UNIX time) */
    __u32 tv_nsec;         /* Nanoseconds since tv_sec */
    __s32 __reserved;      /* Reserved for future uses */
};

```

Figura 5.7: La struttura `statx_timestamp` per i tempi dei file con `statx`.

Anche i campi `stx_atime`, `stx_mtime`, `stx_ctime` mantengono questa analogia, ma esprimono i tempi di ultimo accesso, modifica e cambiamento con una precisione ed estensione maggiore grazie all'uso di una struttura dedicata `statx_timestamp` (riportata in fig. 5.7) che consente di estendere i tempi dei file ad una granularità del nanosecondo e con un valore dello

unix time (vedi sez. 6.4.1) a 64 bit, che non darà problemi di overflow per parecchio tempo (sicuramente ben oltre la durata di questa guida).

Oltre ai precedenti, e a `stx_mask` che abbiamo già visto e che indica quali delle informazioni richieste alla funzione sono state fornite, `statx` prevede una serie di informazioni aggiuntive fornite in altrettanti nuovi campi, illustrati nell'elenco seguente. È comunque previsto che in futuro `statx` venga estesa per supportare ulteriori informazioni.

stx_btime In questo campo viene restituito il *tempo di creazione* del file. Come detto in sez. 4.3.4 questo tempo normalmente non esiste in un sistema *unix-like*, ma per migliorare l'interoperabilità è stato aggiunto nelle versioni più recenti di vari filesystem (come XFS, *ext4*, ecc.) in modo che possa essere utilizzato da servizi di condivisione dei file (è usato da *Samba*, ed è previsto nello standard di NFSv4).

stx_attributes_mask

in questo campo viene restituita una maschera che indica quali sono i bit restituiti in `stx_attributes` effettivamente supportati per il file, e per poter utilizzare quest'ultimo occorre sempre eseguire un AND aritmetico con `stx_attributes_mask` per ottenere i valori validi.

stx_attributes

in questo campo vengono restituiti gli eventuali attributi addizionali posseduti dal file. Gran parte di questi sono quelli impostati con i comandi `lsattr` e `chattr` ed abbiamo già incontrato alcuni di essi in sez. 4.4.1. Gli attributi vengono restituiti in forma di maschera binaria con i valori delle costanti elencate in tab. 5.11, dove si trova anche la relativa descrizione.

Costante	Significato
STATX_ATTR_COMPRESSED	Il file è compresso automaticamente dal filesystem (quindi può richiedere un maggior uso di risorse in caso di accesso).
STATX_ATTR_IMMUTABLE	Il file è marcato come <i>immutable</i> e non può essere modificato in nessun modo (vedi sez. 4.4.1).
STATX_ATTR_APPEND	Il file è marcato come <i>append-only</i> e può essere soltanto esteso in <i>append</i> (vedi sez. 4.4.1).
STATX_ATTR_NODUMP	Il file è marcato per essere escluso da eventuali backup a livello di filesystem come quelli eseguiti con il comando <code>dump</code> .
STATX_ATTR_ENCRYPTED	Il file è cifrato sul filesystem ed è necessaria una chiave di accesso per decifrarne il contenuto.
STATX_ATTR_AUTOMOUNT	Il file, in questo caso in genere una directory, è marcata come punto di innesco per un <i>automount</i> .

Tabella 5.11: Le costanti degli attributi addizionali restituiti in `stx_attributes`.

stx_rdev_major, stx_rdev_minor

in questi campi vengono restituiti rispettivamente *major number* e *minor number* del file quando questo è un file di dispositivo (fanno le veci del campo `st_rdev` di `stat`).

stx_dev_major, stx_dev_minor

in questi campi vengono restituiti *major number* e *minor number* del dispositivo su cui risiede il file (fanno le veci del campo `st_dev` di `stat`).

Di questi campi `stx_mode`, `stx_nlink`, `stx_uid`, `stx_gid`, `stx_ino`, `stx_size` e `stx_blocks` e quelli relativi ai tempi ordinari dei file vengono sempre restituiti, ed il relativo valore in

`statx` sovrascritto, indipendentemente dal fatto che siano stati richiesti o no, con `stx_mask` che indicherà quali sono quelli che hanno valori effettivamente validi.

Se un filesystem ha dei campi che non esistono o hanno valori senza corrispondenza in un sistema unix-like, questi potranno essere restituiti con valori fittizi ricostruiti, ad esempio usando `UID` e `GID` impostati in fase di montaggio per un filesystem che non supporta gli utenti; in questi casi il relativo bit in `stx_mask` sarà comunque cancellato. In caso di cambiamenti al file eseguiti in concorrenza a `statx` è possibile che campi diversi possano avere informazioni ottenute in momenti diversi, con valori precedenti o posteriori al cambiamento. Inoltre, se non richiesti esplicitamente, alcuni campi possono avere valori approssimati, ad esempio in caso di NFS, questi non vengono mai aggiornati dallo stato sul server remoto.

Il campo `stx_btime` viene restituito solo se richiesto, e si otterrà un valore nullo (ed il relativo bit in `stx_mask` cancellato) se questo non esiste. Lo stesso vale nel caso si siano richiesti `stx_rdev_major` o `stx_rdev_minor` ed il file non è un file di dispositivo. I campi `stx_dev_major`, `stx_dev_minor` e `stx_blksize` attengono ad informazioni locali, e sono sempre disponibili in maniera diretta.

Infine trattiamo qui altre due funzioni, `fexecve` e `execveat`, che non attengono che in maniera indiretta all'uso dei file, ma sono comunque legate all'interfaccia delle *at-functions*. In realtà la sola effettivamente collegata all'interfaccia delle *at-functions* è la funzione di sistema `execveat`, introdotta con il kernel 3.19, e per la quale non è disponibile ancora un'interfaccia diretta nella *glibc* che però la usa (quando disponibile) per realizzare `fexecve`.

L'introduzione di queste funzioni nasce dall'esigenza di verificare i contenuti di un file eseguibile prima di eseguirlo. Fare il controllo (aprendo il file e verificandone il contenuto) e poi eseguirlo con `execve` è suscettibile alla possibilità che fra il controllo e l'esecuzione il nome del file o di una directory sovrastante venga cambiato.

Per mitigare il problema nello standard POSIX.1-2008 è stata introdotta la funzione `fexecve` che consente di eseguire un programma usando un file descriptor al posto di un *pathname*; il suo prototipo è:

```
#include <unistd.h>
int fexecve(int fd, char *const argv[], char *const envp[])
                                     Esegue un programma da un file descriptor.
```

La funzione non ritorna in caso di successo e ritorna `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `fd` non è un file descriptor, o `argv` o `envp` sono `NULL`.
`ENOSYS` il filesystem `proc` non è disponibile (prima del kernel 3.19).
oltre a tutti gli errori già visti per `execve`.

La funzione `execve` esegue il programma contenuto nel file (su cui il chiamante deve avere il permesso di esecuzione) corrispondente a `fd`; questo deve essere stato ottenuto aprendo il relativo eseguibile in sola lettura o con `O_PATH`. Questa funzione fino al kernel 3.19 veniva realizzata nella *glibc* usando il filesystem `/proc` per ottenere da `fd` il file corrispondente in `/proc/self/fd/`, in maniera analoga a quanto visto per l'esempio di fig. 5.5.

La funzione di sistema `execveat` è stata introdotta proprio per rendere più sicura l'esecuzione ed evitare la necessità di avere disponibile `/proc` per poter usare `fexecve`, il suo prototipo è:

```
#include <unistd.h>
int execveat(int dirfd, const char *pathname, char *const argv[],
             char *const envp[], int flags)
                                     Esegue un programma relativo ad una directory.
```

La funzione non ritorna in caso di successo e ritorna <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà, inoltre tutti gli errori già visti per <code>execve</code> , uno dei valori:	
<code>EBADF</code>	<code>fd</code> non è un file descriptor valido.
<code>EINVAL</code>	<code>flags</code> non ha un valore valido.
<code>ELOOP</code>	si è usato <code>AT_SYMLINK_NOFOLLOW</code> in <code>flags</code> ma il file indicato è un link simbolico.
<code>ENOENT</code>	il programma di cui si è richiesta l'esecuzione è uno script, ma <code>dirfd</code> è aperto con il flag di <i>close-on-exec</i> e pertanto il programma non sarebbe accessibile all'interprete.

La funzione segue la sintassi delle *at-functions* per indicare il file da eseguire, e per il resto si comporta esattamente con come `execve` (le cui caratteristiche sono già state illustrate in sez. 3.1.6). Diventa così possibile indicare il programma da eseguire sia con un *pathname* assoluto che relativo (usando `AT_FDCWD` come `dirfd`), oppure con un *pathname* relativo alla directory indicata da `dirfd`. In quest'ultima forma l'uso della funzione consente estendere i vantaggi delle *at-functions* anche al caso dell'esecuzione di un programma.

Inoltre usando, per `flags` il valore `AT_EMPTY_PATH`, si può indicare direttamente il file da eseguire aprendolo e passandone il file descriptor nell'argomento `dirfd`, ottenendo il comportamento di `fexecve`; quest'ultima infatti è sostanzialmente equivalente all'esecuzione di:

```
execveat(fd, "", argv, envp, AT_EMPTY_PATH);
```

l'unico altro valore utilizzabile per `flags` è `AT_SYMLINK_NOFOLLOW`, che fa fallire la funzione con un errore di `ELOOP` se il file indicato è un link simbolico.

Quando si usano `execveat` o `fexecve` per eseguire un programma attraverso un file descriptor è naturale impostare sullo stesso il flag di *close-on-exec* in modo che questo venga automaticamente chiuso all'esecuzione. Questo evita di lasciare aperto inutilmente un file descriptor (un programma in genere non ha bisogno di avere un file aperto su se stesso), ma soprattutto evita problemi in caso di un eventuale uso ricorsivo di queste funzioni, in tal caso infatti, restando aperto ad ogni iterazione un ulteriore file descriptor, si potrebbe arrivare all'esaurimento degli stessi.

Tutto questo però non è vero quando si vuole eseguire uno script; in tal caso infatti (si ricordi quanto detto a questo riguardo in sez. 3.1.6) il programma che viene effettivamente messo in esecuzione è l'interprete indicato nella riga iniziale dello script, che poi legge ed interpreta il codice da eseguire dallo script stesso. Ma se lancia lo script usando un file descriptor su cui è attivo il flag di *close-on-exec*, questo sarà già chiuso quando l'interprete viene posto in esecuzione, rendendo impossibile la lettura del programma da interpretare.

Per questo motivo, quando ci si trova in questa situazione, `execveat` (e quindi anche `fexecve`) eseguono un controllo preventivo e falliscono con un errore di `ENOENT`. Pertanto se si vuole eseguire uno script passandone il file descriptor l'unica possibilità è non attivare il flag di *close-on-exec*, esponendosi però al rischio di incorrere nei problemi accennati in precedenza.

5.2.5 Le operazioni di controllo sui file descriptor

Oltre alle operazioni base esaminate in sez. 5.1 esistono tutta una serie di operazioni ausiliarie che è possibile eseguire su un file descriptor, che non riguardano la normale lettura e scrittura di dati, ma la gestione sia delle loro proprietà, che di tutta una serie di ulteriori funzionalità che il kernel può mettere a disposizione.

Per le operazioni di manipolazione e di controllo delle varie proprietà e caratteristiche di un file descriptor, viene usata la funzione di sistema `fcntl`,³⁴ il cui prototipo è:

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd)
int fcntl(int fd, int cmd, int arg)
int fcntl(int fd, int cmd, ...)
```

Esegue una operazione di controllo su un file descriptor.

La funzione ha valori di ritorno diversi a seconda dell'operazione richiesta in caso di successo mentre ritorna sempre `-1` per un errore, nel qual caso `errno` assumerà valori diversi che dipendono dal tipo di operazione, gli unici con significato generico sono:

EBADF `fd` non è un file aperto.

EINVAL `cmd` non è un comando supportato dal kernel corrente.

Il primo argomento della funzione è sempre il numero di file descriptor `fd` su cui si vuole operare. Il comportamento di questa funzione, il numero e il tipo degli argomenti, il valore di ritorno e gli eventuali errori aggiuntivi, sono determinati dal valore del secondo argomento `cmd`, che serve a specificare il “comando” della funzione, in sostanza quale operazione si intende eseguire. A seconda del comando richiesto il terzo argomento può essere assente (ma se specificato lo stesso verrà semplicemente ignorato) ed in generale dipende dal comando `cmd`; il caso più comune è quello di un intero, ma ci sono comandi in cui si devono usare dei tipi specifici, che descriveremo esplicitamente nei singoli casi.

In sez. 5.2.2 abbiamo incontrato un esempio dell'uso di `fcntl` per la duplicazione dei file descriptor, una lista di tutti i possibili valori per `cmd`, e del relativo significato, dei codici di errore restituiti e del tipo del terzo argomento (cui faremo riferimento con il nome indicato nel precedente prototipo), è riportata di seguito:

F_DUPFD trova il primo file descriptor disponibile di valore maggiore o uguale all'argomento `arg`, e ne fa un duplicato di `fd`, ritorna il nuovo file descriptor in caso di successo e `-1` in caso di errore. Oltre a EBADF gli errori possibili sono EINVAL se `arg` è negativo o maggiore del massimo consentito o EMFILE se il processo ha già raggiunto il massimo numero di descrittori consentito.

F_DUPFD_CLOEXEC ha lo stesso effetto di F_DUPFD, ma in più attiva il flag di *close-on-exec* sul file descriptor duplicato, in modo da evitare una successiva chiamata con F_SETFD. La funzionalità è stata introdotta con il kernel 2.6.24 ed è prevista nello standard POSIX.1-2008 (si deve perciò definire `_POSIX_C_SOURCE` ad un valore adeguato secondo quanto visto in sez. 1.3.7).

F_GETFD restituisce il valore dei *file descriptor flags* (vedi sez. 5.2.1) di `fd` in caso di successo o `-1` in caso di errore, il terzo argomento viene ignorato. Non sono previsti errori diversi da EBADF. Al momento l'unico flag usato è quello di *close-on-exec*, identificato dalla costante `FD_CLOEXEC`, che serve a richiedere che il file venga chiuso nella esecuzione di una *exec* (vedi sez. 3.1.6). Un valore nullo significa pertanto che il flag non è impostato.

F_SETFD imposta il valore dei *file descriptor flags* (vedi sez. 5.2.1) al valore specificato con `arg`, ritorna un valore nullo in caso di successo e `-1` in caso di errore. Non sono previsti errori diversi da EBADF. Dato che l'unico flag attualmente usato è quello

³⁴ad esempio si gestiscono con questa funzione varie modalità di I/O asincrono (vedi sez. 10.3) e il *file locking* (vedi sez. 10.1) e altre funzionalità avanzate che tratteremo più avanti.

di *close-on-exec*, identificato dalla costante `FD_CLOEXEC`, tutti gli altri bit di `arg`, anche se impostati, vengono ignorati.³⁵

F_GETFL ritorna il valore dei *file status flags* di `fd` in caso di successo o `-1` in caso di errore, il terzo argomento viene ignorato. Non sono previsti errori diversi da `EBADF`. Il comando permette di rileggere il valore di quei bit dell'argomento `flags` di `open` che vengono memorizzati nella relativa voce della *file table* all'apertura del file, vale a dire quelli riportati in tab. 5.2 e tab. 5.4).

Si ricordi che quando si usa la funzione per determinare le modalità di accesso con cui è stato aperto il file è necessario estrarre i bit corrispondenti nel *file status flag* con la maschera `O_ACCMODE` come già accennato in sez. 5.1.2.

F_SETFL imposta il valore dei *file status flags* al valore specificato da `arg`, ritorna un valore nullo in caso di successo o `-1` in caso di errore. In generale possono essere impostati solo i flag riportati in tab. 5.4, su Linux si possono modificare soltanto `O_APPEND`, `O_ASYNC`, `O_DIRECT`, `O_NOATIME` e `O_NONBLOCK`.

Oltre a `EBADF` si otterrà `EPERM` se si cerca di rimuovere `O_APPEND` da un file marcato come *append-only* o se si cerca di impostare `O_NOATIME` su un file di cui non si è proprietari (e non si hanno i permessi di amministratore) ed `EINVAL` se si cerca di impostare `O_DIRECT` su un file che non supporta questo tipo di operazioni.

F_GETLK richiede un controllo sul *file lock* specificato nella struttura `flock` puntata dal terzo argomento (che pertanto dovrà essere di tipo `struct flock *`) sovrascrivendone il contenuto con il risultato; ritorna un valore nullo in caso di successo o `-1` in caso di errore. Come per i due successivi comandi oltre a `EBADF`, se il terzo argomento non è un puntatore valido, restituisce l'errore generico `EFAULT`. Questa funzionalità è trattata in dettaglio in sez. 10.1.3.

F_SETLK richiede o rilascia un *file lock* a seconda di quanto specificato nella struttura puntata dal terzo argomento (sempre di tipo `struct flock *`); ritorna un valore nullo in caso di successo e `-1` se il *file lock* è tenuto da qualcun altro, nel qual caso si ha un errore di `EACCES` o `EAGAIN`. Questa funzionalità è trattata in dettaglio in sez. 10.1.3.

F_SETLKW identica a `F_SETLK` eccetto per il fatto che la funzione non ritorna subito ma attende che il blocco sia rilasciato, se l'attesa viene interrotta da un segnale la funzione restituisce `-1` e imposta `errno` a `EINTR`. Questa funzionalità è trattata in dettaglio in sez. 10.1.3.

F_OFD_GETLK analoga di `F_GETLK` ma per i nuovi *open file descriptor locks* introdotti con il kernel 3.15, richiede un controllo sul *file lock* specificato nella struttura `flock` puntata dal terzo argomento (che pertanto dovrà essere di tipo `struct flock *`) sovrascrivendone il contenuto con il risultato, ritorna un valore nullo in caso di successo o `-1` in caso di errore. Come per i due successivi comandi oltre a `EBADF` se il terzo argomento non è un puntatore valido restituisce l'errore generico `EFAULT`. Questa funzionalità è trattata in dettaglio in sez. 10.1.4.

F_OFD_SETLK analoga di `F_SETLK` ma per i nuovi *open file descriptor locks* introdotti con il kernel 3.15, richiede o rilascia un *file lock* a seconda di quanto specificato nella

³⁵questo almeno è quanto avviene fino al kernel 3.2, come si può evincere dal codice della funzione `do_fcntl` nel file `fs/fcntl.c` dei sorgenti del kernel.

struttura puntata dal terzo argomento (sempre di tipo `struct flock *`); ritorna un valore nullo in caso di successo e `-1` se il *file lock* è tenuto da qualcun altro, nel qual caso si ha un errore di `EACCES` o `EAGAIN`. Questa funzionalità è trattata in dettaglio in sez. 10.1.4.

`F_OFD_SETLKW` identica a `F_OFD_SETLK` eccetto per il fatto che la funzione non ritorna subito ma attende che il blocco sia rilasciato, se l'attesa viene interrotta da un segnale la funzione restituisce `-1` e imposta `errno` a `EINTR`. Questa funzionalità è trattata in dettaglio in sez. 10.1.4.

`F_GETOWN` restituisce in caso di successo l'identificatore del processo o del *process group* (vedi sez. 8.1.2) che è preposto alla ricezione dei segnali `SIGIO` o `SIGURG`; il primo (o l'eventuale segnale alternativo impostato con `F_SETSIG`) per gli eventi asincroni associati al file descriptor `fd` (vedi sez. 10.3), il secondo per la notifica dei dati urgenti di un socket (vedi sez. 18.1.4). Restituisce `-1` in caso di errore ed il terzo argomento viene ignorato. Non sono previsti errori diversi da `EBADF`.

Per distinguerlo dal caso in cui il segnale viene inviato a un singolo processo, nel caso di un *process group* viene restituito un valore negativo il cui valore assoluto corrisponde all'identificatore del *process group*. Con Linux questo comporta un problema perché se il valore restituito dalla *system call* è compreso nell'intervallo fra `-1` e `-4095` in alcune architetture questo viene trattato dalla *glibc* come un errore,³⁶ per cui in tal caso `fcntl` ritornerà comunque `-1` mentre il valore restituito dalla *system call* verrà assegnato ad `errno`, cambiato di segno.

Per questo motivo con il kernel 2.6.32 è stato introdotto il comando alternativo `F_GETOWN_EX`, che vedremo a breve, che consente di evitare il problema. A partire dalla versione 2.11 la *glibc*, se disponibile, usa questa versione alternativa per mascherare il problema precedente e restituire un valore corretto in tutti i casi.³⁷ Questo però comporta che il comportamento del comando può risultare diverso a seconda delle versioni della *glibc* e del kernel.

`F_SETOWN` imposta, con il valore del terzo argomento `arg`, l'identificatore del processo o del *process group* che riceverà i segnali `SIGIO` e `SIGURG` per gli eventi asincroni associati al file descriptor `fd`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Oltre a `EBADF` gli errori possibili sono `ESRCH` se `arg` indica un processo o un *process group* inesistente.

L'impostazione è soggetta alle stesse restrizioni presenti sulla funzione `kill` (vedi sez. 7.3.3), per cui un utente non privilegiato può inviare i segnali solo ad un processo che gli appartiene, in genere comunque si usa il processo corrente. Come per `F_GETOWN`, per indicare un *process group* si deve usare per `arg` un valore negativo, il cui valore assoluto corrisponda all'identificatore del *process group*.

A partire dal kernel 2.6.12, se si sta operando con i *thread* della implementazione nativa di Linux (quella della NTPL, vedi sez. 12.2.2) e se si è impostato un segnale specifico con `F_SETSIG`, un valore positivo di `arg` viene interpretato come

³⁶il problema deriva dalle limitazioni presenti in architetture come quella dei normali PC (i386) per via delle modalità in cui viene effettuata l'invocazione delle *system call* che non consentono di restituire un adeguato codice di ritorno.

³⁷in cui cioè viene restituito un valore negativo corretto qualunque sia l'identificatore del *process group*, che non potendo avere valore unitario (non esiste infatti un *process group* per `init`) non può generare ambiguità con il codice di errore.

indicante un *Thread ID* e non un *Process ID*. Questo consente di inviare il segnale impostato con `F_SETSIG` ad uno specifico *thread*.

In genere questo non comporta differenze significative per i processi ordinari, in cui non esistono altri *thread*, dato che su Linux il *thread* principale, che in tal caso è anche l'unico, mantiene un valore del *Thread ID* uguale al *PID* del processo. Il problema è però che questo comportamento non si applica a `SIGURG`, per il quale `arg` viene sempre interpretato come l'identificatore di un processo o di un *process group*.

`F_GETOWN_EX` legge nella struttura puntata dal terzo argomento (che deve essere di tipo `struct f_owner_ex *`) l'identificatore del processo, *thread* o *process group* che è preposto alla ricezione dei segnali `SIGIO` e `SIGURG` per gli eventi associati al file descriptor `fd`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Oltre a `EBADF` l'unico altro errore è `EFAULT` se `owner` non è un puntatore valido.

Il comando, che è disponibile solo a partire dal kernel 2.6.32, effettua lo stesso compito di `F_GETOWN` di cui costituisce una evoluzione che consente di superare i limiti e le ambiguità relative ai valori restituiti come identificativo. A partire dalla versione 2.11 della *glibc* esso viene usato dalla libreria per realizzare una versione di `fcntl` che non presenti i problemi illustrati in precedenza per `F_GETOWN`. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

```
struct f_owner_ex {  
    int    type;  
    pid_t  pid;  
};
```

Figura 5.8: La struttura `f_owner_ex`.

`F_SETOWN_EX` imposta con il valore della struttura puntata dal terzo argomento (che deve essere di tipo `struct f_owner_ex *`) l'identificatore del processo o del *process group* che riceverà i segnali `SIGIO` e `SIGURG` per gli eventi associati al file descriptor `fd`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore, con gli stessi errori di `F_SETOWN` più `EINVAL` se il campo `type` di `f_owner_ex` non indica un tipo di identificatore valido.

Come `F_GETOWN_EX` il comando richiede come terzo argomento il puntatore ad una struttura `f_owner_ex` (la cui definizione è riportata in fig. 5.8) in cui il campo `type` indica il tipo di identificatore che si intende usare, mentre il relativo valore è specificato nel campo `pid`, che assume lo stesso significato del terzo argomenti di `F_SETOWN`.

Per `type` i soli valori validi sono `F_OWNER_TID`, `F_OWNER_PID` e `F_OWNER_PGRP`, che indicano rispettivamente che si intende specificare con `pid` un *Tread ID*, un *Process ID* o un *Process Group ID*. A differenza di `F_SETOWN` se si specifica un *Tread ID* questo riceverà sia `SIGIO` (o il segnale impostato con `F_SETSIG`) che `SIGURG`. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 2.6.32 ed è utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

F_GETSIG restituisce in caso di successo il valore del segnale inviato dai vari meccanismi di I/O asincrono associati al file descriptor `fd` (quelli trattati in sez. 10.3) o `-1` in caso di errore, il terzo argomento viene ignorato. Non sono previsti errori diversi da `EBADF`. Un valore nullo indica che si sta usando il segnale predefinito, che è `SIGIO`. Un valore diverso da zero indica il segnale che è stato impostato con `F_SETSIG`, che può essere anche lo stesso `SIGIO`. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

F_SETSIG imposta il segnale inviato dai vari meccanismi di I/O asincrono associati al file descriptor `fd` (quelli trattati in sez. 10.3) al valore indicato da `arg`, ritorna un valore nullo in caso di successo o `-1` in caso di errore. Oltre a `EBADF` gli errori possibili sono `EINVAL` se `arg` indica un numero di segnale non valido. Un valore nullo di `arg` indica di usare il segnale predefinito, cioè `SIGIO`. Un valore diverso da zero, compreso lo stesso `SIGIO`, specifica il segnale voluto. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

L'impostazione di un valore diverso da zero permette inoltre, se si è installato il gestore del segnale come `sa_sigaction` usando `SA_SIGINFO` (vedi sez. 7.4.3), di rendere disponibili al gestore informazioni ulteriori riguardo il file che ha generato il segnale attraverso i valori restituiti in `siginfo_t`. Se inoltre si imposta un segnale *real-time* si potranno sfruttare le caratteristiche di avanzate di questi ultimi (vedi sez. 7.5.1), ed in particolare la capacità di essere accumulati in una coda prima della notifica.

F_GETLEASE restituisce il tipo di *file lease* che il processo detiene nei confronti del file descriptor `fd` o `-1` in caso di errore, il terzo argomento viene ignorato. Non sono previsti errori diversi da `EBADF`. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`. Questa funzionalità è trattata in dettaglio in sez. 10.3.2.

F_SETLEASE imposta o rimuove a seconda del valore del terzo argomento `arg` un *file lease* sul file descriptor `fd`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Oltre a `EBADF` si otterrà `EINVAL` se si è specificato un valore non valido per `arg` (deve essere usato uno dei valori di tab. 10.9), `ENOMEM` se non c'è memoria sufficiente per creare il *file lease*, `EACCES` se non si è il proprietario del file e non si hanno i privilegi di amministratore (per la precisione occorre la capacità `CAP_LEASE`, vedi sez. 9.1.1).

Il supporto per i *file lease*, che consente ad un processo che detiene un *lease* su un file di ricevere una notifica qualora un altro processo cerchi di eseguire una `open` o una `truncate` su di esso, è stato introdotto a partire dai kernel della serie 2.4. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`. Questa funzionalità è trattata in dettaglio in sez. 10.3.2.

F_NOTIFY attiva il meccanismo di notifica asincrona per cui viene riportato al processo chiamante, tramite il segnale `SIGIO` (o altro segnale specificato con `F_SETSIG`) ogni modifica eseguita o direttamente sulla directory cui `fd` fa riferimento, o su uno dei file in essa contenuti; ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux ed utilizzabile solo se si è definita la macro `_GNU_SOURCE`. Questa funzionalità, disponibile dai kernel della serie 2.4.x, è trattata in dettaglio in sez. 10.3.2.

`F_GETPIPE_SZ` restituisce in caso di successo la dimensione del buffer associato alla *pipe* `fd` (vedi sez. 11.1.1) o `-1` in caso di errore, il terzo argomento viene ignorato. Non sono previsti errori diversi da `EBADF`, che viene restituito anche se il file descriptor non è una *pipe*. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 2.6.35, ed è utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

`F_SETPPIPE_SZ` imposta la dimensione del buffer associato alla *pipe* `fd` (vedi sez. 11.1) ad un valore uguale o superiore a quello indicato dall'argomento `arg`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Oltre a `EBADF` gli errori possibili sono `EBUSY` se si cerca di ridurre la dimensione del buffer al di sotto della quantità di dati effettivamente presenti su di esso ed `EPERM` se un processo non privilegiato cerca di impostare un valore troppo alto.

La dimensione minima del buffer è pari ad una pagina di memoria, a cui verrà comunque arrotondata ogni dimensione inferiore, il valore specificato viene in genere arrotondato per eccesso al valore ritenuto più opportuno dal sistema, pertanto una volta eseguita la modifica è opportuno rileggere la nuova dimensione con `F_GETPIPE_SZ`.

I processi non privilegiati (occorre la capacità `CAP_SYS_RESOURCE`) non possono impostare un valore superiore a quello indicato da `/proc/sys/fs/pipe-max-size`. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 2.6.35, ed è utilizzabile solo se si è definita la macro `_GNU_SOURCE`.

`F_GET_SEALS` restituisce in caso di successo l'insieme dei *file seal* presenti su `fd`: 0 se non ve ne sono o `-1` in caso di errore, il terzo argomento viene ignorato. Oltre a `EBADF` se il file non supporta i *file seal* viene restituito un errore di `EINVAL`. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 3.17. Questa funzionalità è trattata in dettaglio in sez. 10.4.5.

`F_ADD_SEALS` aggiunge i *file seal* espressi come maschera binaria nell'argomento `arg` a quelli presenti su `fd`, ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 3.17. Questa funzionalità è trattata in dettaglio in sez. 10.4.5.

`F_GET_RW_HINT` legge il valore dei *read/write hints* associati all'*inode* a cui fa riferimento `fd` nella variabile puntata dal terzo argomento che deve essere di tipo `uint64_t *`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 4.13. Questa funzionalità è trattata in dettaglio in sez. 10.4.4.

`F_SET_RW_HINT` imposta il valore dei *read/write hints* associati all'*inode* a cui fa riferimento `fd`; il valore deve essere fornito nella variabile puntata dal terzo argomento, che deve essere di tipo `uint64_t *`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 4.13. Questa funzionalità è trattata in dettaglio in sez. 10.4.4.

`F_GET_FILE_RW_HINT` legge il valore dei *read/write hints* associati al file descriptor `fd` nella variabile puntata dal terzo argomento che deve essere di tipo `uint64_t *`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 4.13. Questa funzionalità è trattata in dettaglio in sez. 10.4.4.

`F_SET_FILE_RW_HINT` legge il valore dei *read/write hints* associati al file descriptor `fd`; il valore deve essere fornito nella variabile puntata dal terzo argomento, che deve essere di tipo `uint64_t *`. Ritorna un valore nullo in caso di successo o `-1` in caso di errore. Il comando è specifico di Linux, è disponibile solo a partire dal kernel 4.13. Questa funzionalità è trattata in dettaglio in sez. 10.4.4.

La maggior parte delle funzionalità controllate dai comandi di `fcntl` sono avanzate e richiedono degli approfondimenti ulteriori, saranno pertanto riprese più avanti quando affronteremo le problematiche ad esse relative. In particolare le tematiche relative all'I/O asincrono e ai vari meccanismi di notifica saranno trattate in maniera esaustiva in sez. 10.3, quelle relative al *file locking* saranno esaminate in sez. 10.1, quelle relative ai *file seal* in sez. 10.4.5 e quelle relative ai *read/write hints* in sez. 10.4.4. L'uso di questa funzione con i socket verrà trattato in sez. 16.3.

La gran parte dei comandi di `fcntl` (come `F_DUPFD`, `F_GETFD`, `F_SETFD`, `F_GETFL`, `F_SETFL`, `F_GETLK`, `F_SETLK` e `F_SETLKW`) sono previsti da SVr4 e 4.3BSD e standardizzati in POSIX.1-2001 che inoltre prevede gli ulteriori `F_GETOWN` e `F_SETOWN`. Pertanto nell'elenco si sono indicate esplicitamente soltanto le ulteriori richieste in termini delle macro di funzionalità di sez. 1.3.7 soltanto per le funzionalità inserite in standard successivi o specifiche di Linux.

Benché l'interfaccia di gestione dell'I/O sui file di cui abbiamo parlato finora si sia dimostrata valida anche per l'interazione diretta con le periferiche attraverso i loro file di dispositivo, consentendo di usare le stesse funzioni utilizzate per i normali file di dati, esistono però caratteristiche peculiari, specifiche dell'hardware e delle funzionalità che ciascun dispositivo può provvedere, che non possono venire comprese in questa interfaccia astratta come ad esempio l'impostazione della velocità di una porta seriale, o le dimensioni di un framebuffer.

Per questo motivo nell'architettura del sistema è stata prevista l'esistenza di una apposita funzione di sistema, `ioctl`, come meccanismo generico per compiere operazioni specialistiche; il suo prototipo è:

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ...)
```

Esegue una operazione speciale.

La funzione ritorna 0 in caso di successo nella maggior parte dei casi, ma alcune operazioni possono restituire un valore positivo, mentre ritorna sempre `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` gli argomenti `request` o `argp` non sono validi.

`ENOTTY` il file `fd` non è associato con un dispositivo, o la richiesta non è applicabile all'oggetto a cui fa riferimento `fd`.

ed inoltre `EBADF` e `EFAULT` nel loro significato generico.

La funzione richiede che si passi come primo argomento un file descriptor `fd` regolarmente aperto, mentre l'operazione da compiere deve essere indicata dal valore dell'argomento `request`. Il terzo argomento dipende dall'operazione prescelta; tradizionalmente è specificato come `char * argp`, da intendersi come puntatore ad un area di memoria generica (all'epoca della creazione di questa funzione infatti ancora non era stato introdotto il tipo `void`) ma per certe operazioni può essere omesso, e per altre è un semplice intero.

Normalmente la funzione ritorna zero in caso di successo e `-1` in caso di errore, ma per alcune operazioni il valore di ritorno, che nel caso viene impostato ad un valore positivo, può essere utilizzato come indicazione del risultato della stessa. È più comune comunque restituire i risultati all'indirizzo puntato dal terzo argomento.

Data la genericità dell'interfaccia non è possibile classificare in maniera sistematica le operazioni che si possono gestire con `ioctl`, un breve elenco di alcuni esempi di esse è il seguente:

- il cambiamento dei font di un terminale.
- l'esecuzione di una traccia audio di un CD.
- i comandi di avanti veloce e di riavvolgimento di un nastro.
- il comando di espulsione di un dispositivo rimovibile.
- l'impostazione della velocità trasmissione di una linea seriale.
- l'impostazione della frequenza e della durata dei suoni emessi dallo speaker.
- l'impostazione degli attributi dei file (vedi sez. 4.4.3) su un filesystem.³⁸

In generale ogni dispositivo ha un suo insieme di operazioni specifiche effettuabili attraverso `ioctl`, tutte queste sono definite nell'header file `sys/ioctl.h`, e devono essere usate solo sui dispositivi cui fanno riferimento. Infatti anche se in genere i valori di `request` sono opportunamente differenziati a seconda del dispositivo³⁹ così che la richiesta di operazioni relative ad altri dispositivi usualmente provoca il ritorno della funzione con una condizione di errore, in alcuni casi, relativi a valori assegnati prima che questa differenziazione diventasse pratica corrente, si potrebbero usare valori validi anche per il dispositivo corrente, con effetti imprevedibili o indesiderati.

Data la assoluta specificità della funzione, il cui comportamento varia da dispositivo a dispositivo, non è possibile fare altro che dare una descrizione sommaria delle sue caratteristiche; torneremo ad esaminare in seguito quelle relative ad alcuni casi specifici, ad esempio la gestione dei terminali è effettuata attraverso `ioctl` in quasi tutte le implementazioni di Unix, mentre per l'uso di `ioctl` con i socket si veda sez. 16.3.

Riportiamo qui solo l'elenco delle operazioni che sono predefinite per qualunque file, caratterizzate dal prefisso `FIO`. Queste operazioni sono definite nel kernel a livello generale, e vengono sempre interpretate per prime, per cui, come illustrato in [?], eventuali operazioni specifiche che usino lo stesso valore verrebbero ignorate:

<code>FIOCLEX</code>	imposta il flag di <i>close-on-exec</i> sul file, in questo caso, essendo usata come operazione logica, <code>ioctl</code> non richiede un terzo argomento, il cui eventuale valore viene ignorato.
<code>FIONCLEX</code>	cancella il flag di <i>close-on-exec</i> sul file, in questo caso, essendo usata come operazione logica, <code>ioctl</code> non richiede un terzo argomento, il cui eventuale valore viene ignorato.
<code>FIOASYNC</code>	abilita o disabilita la modalità di I/O asincrono sul file (vedi sez. 10.3.1); il terzo argomento deve essere un puntatore ad un intero (cioè di tipo <code>const int *</code>) che contiene un valore logico (un valore nullo disabilita, un valore non nullo abilita).
<code>FIONBIO</code>	abilita o disabilita sul file l'I/O in modalità non bloccante; il terzo argomento deve essere un puntatore ad un intero (cioè di tipo <code>const int *</code>) che contiene un valore logico (un valore nullo disabilita, un valore non nullo abilita).

³⁸i comandi `lsattr` e `chattr` fanno questo con delle `ioctl` dedicate, usabili solo sui filesystem che li supportano.

³⁹il kernel usa un apposito *magic number* per distinguere ciascun dispositivo nella definizione delle macro da usare per `request`, in modo da essere sicuri che essi siano sempre diversi, ed il loro uso per dispositivi diversi causi al più un errore. Si veda il capitolo quinto di [?] per una trattazione dettagliata dell'argomento.

- FIOSETOWN** imposta il processo che riceverà i segnali SIGURG e SIGIO generati sul file; il terzo argomento deve essere un puntatore ad un intero (cioè di tipo `const int *`) il cui valore specifica il PID del processo.
- FIOGETOWN** legge il processo che riceverà i segnali SIGURG e SIGIO generati sul file; il terzo argomento deve essere un puntatore ad un intero (cioè di tipo `int *`) su cui sarà scritto il PID del processo.
- FIONREAD** legge il numero di byte disponibili in lettura sul file descriptor; questa operazione è disponibile solo su alcuni file descriptor, in particolare sui socket (vedi sez. 16.3.3) o sui file descriptor di *epoll* (vedi sez. 10.2.4), il terzo argomento deve essere un puntatore ad un intero (cioè di tipo `int *`) su cui sarà restituito il valore.
- FIOQSIZE** restituisce la dimensione corrente di un file o di una directory, mentre se applicata ad un dispositivo fallisce con un errore di ENOTTY; il terzo argomento deve essere un puntatore ad un intero (cioè di tipo `int *`) su cui sarà restituito il valore.

Si noti però come la gran parte di queste operazioni specifiche dei file (per essere precisi le prime sei dell'elenco) siano effettuabili in maniera generica anche tramite l'uso di `fcntl`. Le due funzioni infatti sono molto simili e la presenza di questa sovrapposizione è principalmente dovuta al fatto che alle origini di Unix i progettisti considerarono che era necessario trattare diversamente rispetto alle operazione di controllo delle modalità di I/O file e dispositivi usando `fcntl` per i primi e `ioctl` per i secondi, all'epoca tra l'altro i dispositivi che usavano `ioctl` erano sostanzialmente solo i terminali, il che spiega l'uso comune di ENOTTY come codice di errore. Oggi non è più così ma le due funzioni sono rimaste.

5.3 L'interfaccia standard ANSI C

Come visto in sez. 5.1 le operazioni di I/O sui file sono gestibili a basso livello con l'interfaccia standard unix, che ricorre direttamente alle *system call* messe a disposizione dal kernel.

Questa interfaccia però non provvede le funzionalità previste dallo standard ANSI C, che invece sono realizzate attraverso opportune funzioni di libreria. Queste funzioni di libreria, insieme alle altre funzioni definite dallo standard (che sono state implementate la prima volta da Ritchie nel 1976 e da allora sono rimaste sostanzialmente immutate), vengono a costituire il nucleo della *glibc* per la gestione dei file.

Esamineremo in questa sezione le funzioni base di questa interfaccia che chiameremo, per distinguerla dalla precedente “degli *stream*”. Esse sono analoghe a quelle di sez. 5.1 per i file descriptor, ed in particolare vedremo come aprire, leggere, scrivere e cambiare la posizione corrente in uno *stream*.

5.3.1 I file *stream*

Come più volte ribadito, l'interfaccia dei file descriptor è un'interfaccia di basso livello, che non provvede nessuna forma di formattazione dei dati e nessuna forma di bufferizzazione per ottimizzare le operazioni di I/O.

In [?] Stevens descrive una serie di test sull'influenza delle dimensioni del blocco di dati (l'argomento `buf` di `read` e `write`) nell'efficienza nelle operazioni di I/O con i file descriptor, evidenziando come le prestazioni ottimali si ottengano a partire da dimensioni del buffer dei

dati pari a quelle dei blocchi del filesystem (il valore dato dal campo `st_blksize` di `stat`), che di norma corrispondono alle dimensioni dei settori fisici in cui è suddiviso il disco.

Se il programmatore non si cura di effettuare le operazioni in blocchi di dimensioni adeguate, le prestazioni sono inferiori. La caratteristica principale dell'interfaccia degli *stream* è che essa provvede da sola alla gestione dei dettagli della bufferizzazione e all'esecuzione delle operazioni di lettura e scrittura in blocchi di dimensioni appropriate all'ottenimento della massima efficienza.

Per questo motivo l'interfaccia viene chiamata anche interfaccia dei *file stream*, dato che non è più necessario doversi preoccupare dei dettagli con cui viene gestita la comunicazione con l'hardware sottostante (come nel caso della dimensione dei blocchi del filesystem), ed un file può essere sempre considerato come composto da un flusso continuo di dati, da cui deriva appunto il nome *stream*.

A parte i dettagli legati alla gestione delle operazioni di lettura e scrittura, sia per quel che riguarda la bufferizzazione che le formattazioni, per tutto il resto i *file stream* restano del tutto equivalenti ai file descriptor (sui quali sono basati), ed in particolare continua a valere quanto visto in sez. 5.2.1 a proposito dell'accesso concorrente ed in sez. 4.4 per il controllo di accesso.

Per ragioni storiche la struttura di dati che rappresenta uno *stream* è stata chiamata `FILE`, questi oggetti sono creati dalle funzioni di libreria e contengono tutte le informazioni necessarie a gestire le operazioni sugli *stream*, come la posizione corrente, lo stato del buffer e degli indicatori di stato e di fine del file.

Per questo motivo gli utenti non devono mai utilizzare direttamente o allocare queste strutture (che sono dei *tipi opachi*) ma usare sempre puntatori del tipo `FILE *` ottenuti dalla libreria stessa, tanto che in certi casi il termine di puntatore a file è diventato sinonimo di *stream*. Tutte le funzioni della libreria che operano sui file accettano come argomenti solo variabili di questo tipo, che diventa accessibile includendo l'header file `stdio.h`.

Ai tre file descriptor standard (vedi tab. 5.1) aperti per ogni processo, corrispondono altrettanti *stream*, che rappresentano i canali standard di input/output prestabiliti; anche questi tre *stream* sono identificabili attraverso dei nomi simbolici definiti nell'header `stdio.h` che sono:

<code>FILE *stdin</code>	Lo <i>standard input</i> cioè il <i>file stream</i> da cui il processo riceve ordinariamente i dati in ingresso. Normalmente è associato dalla shell all'input del terminale e prende i caratteri dalla tastiera.
<code>FILE *stdout</code>	Lo <i>standard output</i> cioè il <i>file stream</i> su cui il processo invia ordinariamente i dati in uscita. Normalmente è associato dalla shell all'output del terminale e scrive sullo schermo.
<code>FILE *stderr</code>	Lo <i>standard error</i> cioè il <i>file stream</i> su cui il processo è supposto inviare i messaggi di errore. Normalmente anch'esso è associato dalla shell all'output del terminale e scrive sullo schermo.

Nella *glibc* `stdin`, `stdout` e `stderr` sono effettivamente tre variabili di tipo `FILE *` che possono essere usate come tutte le altre, ad esempio si può effettuare una redirectione dell'output di un programma con il semplice codice:

```
fclose(stdout);  
stdout = fopen("standard-output-file", "w");
```

ma in altri sistemi queste variabili possono essere definite da macro, e se si hanno problemi di portabilità e si vuole essere sicuri, diventa opportuno usare la funzione `freopen`.

5.3.2 Le modalità di bufferizzazione

La bufferizzazione è una delle caratteristiche principali dell'interfaccia degli *stream*; lo scopo è quello di ridurre al minimo il numero di *system call* (*read* o *write*) eseguite nelle operazioni di input/output. Questa funzionalità è assicurata automaticamente dalla libreria, ma costituisce anche uno degli aspetti più comunemente fraintesi, in particolare per quello che riguarda l'aspetto della scrittura dei dati sul file.

I dati che vengono scritti su di uno *stream* normalmente vengono accumulati in un buffer e poi trasmessi in blocco, con l'operazione che viene usualmente chiamata *scaricamento* del buffer (dal termine inglese *flush*) tutte le volte che questo viene riempito. Questa operazione avviene perciò in maniera asincrona rispetto alla scrittura. Un comportamento analogo avviene anche in lettura (cioè dal file viene letto un blocco di dati, anche se ne sono richiesti una quantità inferiore), ma la cosa ovviamente ha rilevanza inferiore, dato che i dati letti sono sempre gli stessi. In caso di scrittura invece, quando si ha un accesso contemporaneo allo stesso file (ad esempio da parte di un altro processo) si potranno vedere solo le parti effettivamente scritte, e non quelle ancora presenti nel buffer.

Per lo stesso motivo, in tutte le situazioni in cui si sta facendo dell'input/output interattivo, bisognerà tenere presente le caratteristiche delle operazioni di scaricamento dei dati, poiché non è detto che ad una scrittura sullo *stream* corrisponda una immediata scrittura sul dispositivo, e la cosa è particolarmente evidente con le operazioni di input/output sul terminale.

Per rispondere ad esigenze diverse lo standard definisce tre distinte modalità in cui può essere eseguita la bufferizzazione, delle quali occorre essere ben consapevoli, specie in caso di lettura e scrittura da dispositivi interattivi:

- *unbuffered*: in questo caso non c'è bufferizzazione ed i caratteri vengono trasmessi direttamente al file non appena possibile (effettuando immediatamente una *write*);
- *line buffered*: in questo caso i caratteri vengono normalmente trasmessi al file in blocco ogni volta che viene incontrato un carattere di *newline* (il carattere ASCII `\n`) cioè un a capo (in sostanza quando si preme invio);
- *fully buffered*: in questo caso i caratteri vengono trasmessi da e verso il file in blocchi di dimensione opportuna.

Lo standard ANSI C specifica inoltre che lo *standard output* e lo *standard input* siano aperti in modalità *fully buffered* quando non fanno riferimento ad un dispositivo interattivo, e che lo *standard error* non sia mai aperto in modalità *fully buffered*.

Linux, come BSD e SVr4, specifica il comportamento predefinito in maniera ancora più precisa, e cioè impone che lo *standard error* sia sempre *unbuffered*, in modo che i messaggi di errore siano mostrati il più rapidamente possibile, e che *standard input* *standard output* siano aperti in modalità *line buffered* quando sono associati ad un terminale (od altro dispositivo interattivo) ed in modalità *fully buffered* altrimenti.

Il comportamento specificato per *standard input* e *standard output* vale anche per tutti i nuovi *stream* aperti da un processo; la selezione comunque avviene automaticamente, e la libreria apre lo *stream* nella modalità più opportuna a seconda del file o del dispositivo scelto.

La modalità *line buffered* è quella che necessita di maggiori chiarimenti e attenzioni per quel che concerne il suo funzionamento. Come già accennato nella descrizione, di *norma* i dati vengono inviati al kernel alla ricezione di un carattere di *a capo* (il *newline*); questo non è vero in tutti i casi, infatti, dato che le dimensioni del buffer usato dalle librerie sono fisse, se le si eccedono si può avere uno scarico dei dati anche prima che sia stato inviato un carattere di *newline*.

Un secondo punto da tenere presente, particolarmente quando si ha a che fare con I/O interattivo, è che quando si effettua una lettura da uno *stream* che comporta l'accesso alle *system call* del kernel, ad esempio se lo *stream* da cui si legge è in modalità *unbuffered*, viene anche eseguito lo scarico di tutti i buffer degli *stream* in scrittura. In sez. 5.4.2 vedremo come la libreria definisca delle opportune funzioni per controllare le modalità di bufferizzazione e lo scarico dei dati.

5.3.3 Apertura e chiusura di uno *stream*

Le funzioni che si possono usare per aprire uno *stream* sono solo tre: `fopen`, `fdopen` e `freopen`,⁴⁰ ed i rispettivi prototipi sono:

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode)
FILE *fdopen(int fildes, const char *mode)
FILE *freopen(const char *path, const char *mode, FILE *stream)
```

Apre uno *stream* da un *pathname*.
 Associa uno *stream* a un file descriptor.
 Chiude uno *stream* e lo riapre su un file diverso.

Le funzioni ritornano un puntatore ad un oggetto `FILE` in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà il valore ricevuto dalla funzione sottostante di cui è fallita l'esecuzione, gli errori pertanto possono essere quelli di `malloc` per tutte e tre le funzioni, quelli `open` per `fopen`, quelli di `fcntl` per `fdopen` e quelli di `fopen`, `fclose` e `fflush` per `freopen`.

Normalmente la funzione che si usa per aprire uno *stream* è `fopen`, essa apre il file specificato dal *pathname* `path` nella modalità specificata da `mode`, che è una stringa che deve iniziare con almeno uno dei valori indicati in tab. 5.12, anche se sono possibili varie estensioni che vedremo in seguito.

L'uso più comune di `freopen` è per redirigere uno dei tre file standard (vedi sez. 5.3.1): il file `path` viene aperto nella modalità indicata da `mode` ed associato allo *stream* indicato dall'argomento `stream`, e se questo era uno *stream* già aperto esso viene preventivamente chiuso e tutti i dati pendenti vengono scaricati.

Infine `fdopen` viene usata per associare uno *stream* ad un file descriptor esistente ottenuto tramite una altra funzione (ad esempio con una `open`, una `dup`, o una `pipe`) e serve quando si vogliono usare gli *stream* con file come le *fifo* o i socket, che non possono essere aperti con le funzioni delle librerie standard del C.

In realtà lo standard ANSI C prevede un totale di 15 possibili valori diversi per `mode`, ma in tab. 5.12 si sono riportati solo i sei valori effettivi, ad essi può essere aggiunto pure il carattere "b" (come ultimo carattere o nel mezzo agli altri per le stringhe di due caratteri) che in altri sistemi operativi serve a distinguere i file binari dai file di testo; in un sistema POSIX questa distinzione non esiste e il valore viene accettato solo per compatibilità, ma non ha alcun effetto.

La *glibc* supporta alcune estensioni, queste devono essere sempre indicate dopo aver specificato il `mode` con uno dei valori della seconda sezione di tab. 5.12. Ad esempio l'uso del carattere "e" serve ad impostare il *close-on-exec* sul file (è analoga all'uso del flag `O_CLOEXEC` in `open`), "x" serve per evitare di sovrascrivere un file già esistente (è analoga all'uso del flag `O_EXCL` in `open`): se il file specificato già esiste e si aggiunge questo carattere a `mode` la `fopen` fallisce.

Altri due valori hanno usi specialistici, con "m" si chiede di usare il *memory mapping* per l'accesso al file (tratteremo i file mappati in memoria in sez. 10.4.1), ma la funzionalità è

⁴⁰`fopen` e `freopen` fanno parte dello standard ANSI C, `fdopen` è parte dello standard POSIX.1.

Valore	Significato
r	Il file viene aperto, l'accesso viene posto in sola lettura, lo <i>stream</i> è posizionato all'inizio del file.
r+	Il file viene aperto, l'accesso viene posto in lettura e scrittura, lo <i>stream</i> è posizionato all'inizio del file.
w	Il file viene aperto e troncato a lunghezza nulla (o creato se non esiste), l'accesso viene posto in sola scrittura, lo <i>stream</i> è posizionato all'inizio del file.
w+	Il file viene aperto e troncato a lunghezza nulla (o creato se non esiste), l'accesso viene posto in scrittura e lettura, lo <i>stream</i> è posizionato all'inizio del file.
a	Il file viene aperto (o creato se non esiste) in <i>append mode</i> , l'accesso viene posto in sola scrittura.
a+	Il file viene aperto (o creato se non esiste) in <i>append mode</i> , l'accesso viene posto in lettura e scrittura.
b	Specifica che il file è binario, non ha alcun effetto.
c	Evita che l'apertura e seguenti letture o scritture diventino un <i>cancellation point</i> per i <i>thread</i> ; presente dalla <i>glibc</i> 2.3.3.
e	Apri il file con il flag di <code>O_CLOEXEC</code> ; presente dalla <i>glibc</i> 2.7.
m	Cerca di accedere al file con <code>mmap</code> invece che con le funzioni di I/O classiche; presente dalla <i>glibc</i> 2.3.
x	L'apertura fallisce se il file esiste già (ignorato con <code>fdopen</code>).

Tabella 5.12: Modalità di apertura di uno *stream* dello standard ANSI C che sono sempre presenti in qualunque sistema POSIX.

al momento disponibile solo per i file aperti in sola lettura. Con “c” infine si richiede che l'apertura, e le successive operazioni di lettura e scrittura, non diventino un *cancellation point* per i *thread* (tratteremo l'argomento in sez. ??).

Un'altra estensione serve a supportare la localizzazione, quando si aggiunge a *mode* una stringa della forma “,ccs=STRING” (che deve essere sempre in coda a tutte le altre) il valore *STRING* è considerato il nome di una codifica dei caratteri e `fopen` marca il file per l'uso dei caratteri estesi e abilita le opportune funzioni di conversione in lettura e scrittura.

Nel caso si usi `fdopen` i valori specificati da *mode* devono essere compatibili con quelli con cui il file descriptor è stato aperto. Inoltre i modi *w* e *w+* non troncano il file. La posizione nello *stream* viene impostata a quella corrente nel file descriptor, e le variabili di errore e di fine del file (vedi sez. 5.3.4) sono cancellate. Il file non viene duplicato e verrà chiuso automaticamente alla chiusura dello *stream*.

I nuovi file saranno creati secondo quanto visto in sez. 4.4.4 ed avranno i permessi di accesso impostati al valore `S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH` (pari a `0666`) modificato secondo il valore della *umask* per il processo (si veda sez. 4.4.3). Una volta aperto lo *stream*, si può cambiare la modalità di bufferizzazione (si veda sez. 5.4.2) fintanto che non si è effettuato alcuna operazione di I/O sul file.

In caso di file aperti in lettura e scrittura occorre ricordarsi che c'è di mezzo una bufferizzazione; per questo motivo lo standard ANSI C richiede che ci sia un'operazione di posizionamento fra un'operazione di output ed una di input o viceversa (eccetto il caso in cui l'input ha incontrato la fine del file), altrimenti una lettura può ritornare anche il risultato di scritture precedenti l'ultima effettuata.

Per questo motivo è una buona pratica (e talvolta necessario) far seguire ad una scrittura una delle funzioni `fflush`, `fseek`, `fsetpos` o `rewind` prima di eseguire una riletture; viceversa nel caso in cui si voglia fare una scrittura subito dopo aver eseguito una lettura occorre prima usare una delle funzioni `fseek`, `fsetpos` o `rewind`. Anche un'operazione nominalmente nulla come `fseek(file, 0, SEEK_CUR)` è sufficiente a garantire la sincronizzazione.

Una volta completate le operazioni su di esso uno *stream* può essere chiuso con la funzione `fclose`, il cui prototipo è:

```
#include <stdio.h>
int fclose(FILE *stream)
```

Chiude uno *stream*.

La funzione ritorna 0 in caso di successo e EOF per un errore, nel qual caso `errno` assumerà il valore `EBADF` se il file descriptor indicato da *stream* non è valido, o uno dei valori specificati dalla sottostante funzione che è fallita (`close`, `write` o `fflush`).

La funzione chiude lo *stream* *stream* ed effettua lo scarico di tutti i dati presenti nei buffer di uscita e scarta tutti i dati in ingresso; se era stato allocato un buffer per lo *stream* questo verrà rilasciato. La funzione effettua lo scarico solo per i dati presenti nei buffer in *user space* usati dalla *glibc*; se si vuole essere sicuri che il kernel forzi la scrittura su disco occorrerà effettuare una *sync* (vedi sez. 5.2.3).

Linux supporta anche un'altra funzione, `fcloseall`, come estensione GNU implementata dalla *glibc*, accessibile avendo definito `_GNU_SOURCE`, il suo prototipo è:

```
#include <stdio.h>
int fcloseall(void)
```

Chiude tutti gli *stream*.

La funzione ritorna 0 in caso di successo e EOF per un errore, nel qual caso `errno` assumerà gli stessi valori di `fclose`.

La funzione esegue lo scarico dei dati bufferizzati in uscita e scarta quelli in ingresso, chiudendo tutti i file. Questa funzione è provvista solo per i casi di emergenza, quando si è verificato un errore ed il programma deve essere abortito, ma si vuole compiere qualche altra operazione dopo aver chiuso i file e prima di uscire (si ricordi quanto visto in sez. 2.1.3).

5.3.4 Gestione dell'I/O e posizionamento su uno *stream*

Una delle caratteristiche più utili dell'interfaccia degli *stream* è la ricchezza delle funzioni disponibili per le operazioni di lettura e scrittura sui file. Sono infatti previste ben tre diverse modalità di input/output non formattato:

- *Input/Output binario*, una modalità in cui si leggono e scrivono blocchi di dati di dimensione arbitraria; è l'analogo della modalità ordinaria dell'input/output sui file descriptor vista in sez. 5.1.4 e 5.1.5.
- *Input/Output a caratteri*, una modalità in cui si legge e scrive un singolo carattere alla volta, anche in questo caso la bufferizzazione viene gestita automaticamente dalla libreria;
- *Input/Output di linea*, una modalità in cui si legge e scrive una linea di testo alla volta, in questa modalità si intende per linea una sequenza di caratteri terminata dal carattere di *newline* (`'\n'`);

A queste tre modalità si aggiunge poi la modalità di input/output formattato che tratteremo in sez. 5.3.5. Ognuna di queste modalità utilizza per l'I/O delle funzioni specifiche che vedremo più avanti, affronteremo qui invece gli argomenti e le funzioni che si applicano in generale a tutte queste diverse modalità di I/O.

Una prima caratteristica specifica è che differenza di quanto avviene con l'interfaccia dei file descriptor, con gli *stream* il raggiungimento della fine del file viene considerato un errore, che viene notificato come tale dai valori di uscita delle varie funzioni.

In vari casi questo avviene con la restituzione di uno specifico valore intero (di tipo `int`) definito come EOF nell'header `stdlib.h`. La costante deve essere negativa perché in molte funzioni un valore positivo indica la quantità di dati scritti e ci potrebbe essere sovrapposizione, la *glibc* usa il valore `-1`, ma altre implementazioni possono avere valori diversi.

Dato che le funzioni dell'interfaccia degli *stream* sono funzioni di libreria realizzate usando delle *system call*, esse non modificano mai direttamente la variabile `errno`, che in caso di errore mantiene sempre il valore impostato dalla *system call* sottostante che lo ha riportato.

Siccome la condizione di *end-of-file* è anch'essa segnalata come errore, nasce il problema di come distinguerla; basarsi solo sul valore di ritorno della funzione e controllare il valore di `errno` infatti non basta, dato che quest'ultimo potrebbe essere stato impostato in una altra occasione, (si veda sez. 6.5.1 per i dettagli del funzionamento di `errno`).

Per questo motivo tutte le implementazioni delle librerie standard mantengono per ogni *stream* almeno due flag all'interno dell'oggetto `FILE`, il flag di *end-of-file*, che segnala che si è raggiunta la fine del file in lettura, e quello di errore, che segnala la presenza di un qualche errore nelle operazioni di input/output; questi due flag possono essere riletti dalle funzioni `feof` e `ferror`, i cui prototipi sono:

<pre>#include <stdio.h> int feof(FILE *stream)</pre>	Controlla il flag di <i>end-of-file</i> di uno <i>stream</i> .
<pre>int ferror(FILE *stream)</pre>	Controlla il flag di errore di uno <i>stream</i> .
Le funzioni ritornano un valore diverso da zero se i relativi flag sono impostati, e non prevedono condizioni di errore.	

Si tenga presente comunque che la lettura di questi flag segnala soltanto che c'è stato un errore o che si è raggiunta la fine del file in una qualunque operazione sullo *stream*, il controllo su quanto avvenuto deve quindi essere effettuato ogni volta che si chiama una funzione di libreria.

Entrambi i flag (di errore e di *end-of-file*) possono essere cancellati usando la funzione `clearerr`, il cui prototipo è:

<pre>#include <stdio.h> void clearerr(FILE *stream)</pre>	Cancella i flag di errore ed <i>end-of-file</i> di uno <i>stream</i> .
La funzione non ritorna nulla e prevede condizioni di errore.	

In genere si usa questa funzione una volta che si sia identificata e corretta la causa di un errore per evitare di mantenere i flag attivi, così da poter rilevare una successiva ulteriore condizione di errore. Di questa funzione esiste una analoga `clearerr_unlocked` (con lo stesso argomento e stessi valori di ritorno) che non esegue il blocco dello *stream* (tratteremo il significato di blocco di uno *stream* in sez. 5.4.3).

Come per i file descriptor anche per gli *stream* è possibile spostarsi all'interno di un file per effettuare operazioni di lettura o scrittura in un punto prestabilito, sempre che l'operazione di riposizionamento sia supportata dal file sottostante lo *stream*, nel caso cioè in cui si ha a che fare con quello che viene detto un file ad *accesso casuale*. Dato che in un sistema Unix esistono vari tipi di file, come le *fifo* ed i file di dispositivo (ad esempio i terminali), non è scontato che questo sia vero in generale, pur essendolo sempre nel caso di file di dati.

Con Linux ed in generale in ogni sistema unix-like la posizione nel file, come abbiamo già visto in sez. 5.1.3, è espressa da un intero positivo, rappresentato dal tipo `off_t`. Il problema è che alcune delle funzioni usate per il riposizionamento sugli *stream* originano dalle prime versioni di Unix, in cui questo tipo non era ancora stato definito, e che in altri sistemi non

è detto che la posizione su un file venga sempre rappresentata con il numero di caratteri dall'inizio: ad esempio nel VMS dove esistono i file a record (in cui cioè l'I/O avviene per blocchi, i record, di dimensione fissa), essa può essere rappresentata come un numero di record, più l'offset rispetto al record corrente.

Tutto questo comporta la presenza di diverse funzioni che eseguono sostanzialmente le stesse operazioni, ma usano argomenti di tipo diverso. Le funzioni tradizionali usate per eseguire una modifica della posizione corrente sul file con uno *stream* sono *fseek* e *rewind*, i rispettivi prototipi sono:

<pre>#include <stdio.h> int fseek(FILE *stream, long offset, int whence) void rewind(FILE *stream)</pre>	<p>Sposta la posizione nello <i>stream</i>.</p> <p>Riporta la posizione nello <i>stream</i> all'inizio del file.</p>
---	--

La funzione <i>fseek</i> ritorna 0 in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà i valori di <i>lseek</i> , <i>rewind</i> non ritorna nulla e non ha condizioni di errore.
--

L'uso di *fseek* è del tutto analogo a quello di *lseek* per i file descriptor (vedi sez. 5.1.3). Anche gli argomenti, a parte il tipo, hanno esattamente lo stesso significato. In particolare *whence* deve assumere gli stessi valori già visti nella prima parte di tab. 5.5. La funzione restituisce 0 in caso di successo e -1 in caso di errore.

La funzione *rewind* riporta semplicemente la posizione corrente sul file all'inizio dello *stream*, ma non è esattamente equivalente ad aver eseguito una *fseek(stream, 0L, SEEK_SET)*, in quanto con l'uso della funzione vengono cancellati anche i flag di errore e di fine del file.

Per ottenere la posizione corrente sul file di uno *stream* lo standard ANSI C prescrive l'uso della funzione *ftell*, il cui prototipo è:

<pre>#include <stdio.h> long ftell(FILE *stream)</pre>	<p>Legge la posizione attuale nello <i>stream</i>.</p>
--	--

La funzione ritorna la posizione corrente in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà i valori di <i>lseek</i> .
--

che restituisce la posizione come numero di byte dall'inizio dello *stream*.

Sia *fseek* che *ftell* esprimono la posizione nel file con un intero di tipo *long*. Dato che in certi casi, ad esempio quando si usa un filesystem indicizzato a 64 bit su una macchina con architettura a 32 bit, questo può non essere possibile lo standard POSIX ha introdotto le nuove funzioni *fgetpos* e *fsetpos*, che invece usano il nuovo tipo *fpos_t*, ed i cui prototipi sono:

<pre>#include <stdio.h> int fsetpos(FILE *stream, fpos_t *pos) int fgetpos(FILE *stream, fpos_t *pos)</pre>	<p>Imposta la posizione corrente sul file.</p> <p>Legge la posizione corrente sul file.</p>
--	---

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <i>errno</i> assumerà i valori di <i>lseek</i> .
--

In Linux, a partire dalle glibc 2.1, sono presenti anche le due funzioni *fseeko* e *ftello*, che sono assolutamente identiche alle precedenti *fseek* e *ftell* ma hanno argomenti di tipo *off_t* anziché di tipo *long int*. Dato che *long* è nella gran parte dei casi un intero a 32 bit, questo diventa un problema quando la posizione sul file viene espressa con un valore a 64 bit come accade nei sistemi più moderni.

5.3.5 Input/output non formattato

La prima modalità di input/output non formattato ricalca quella della interfaccia dei file descriptor, e provvede semplicemente la scrittura e la lettura dei dati da un buffer verso un file e viceversa. In generale questa è la modalità che si usa quando si ha a che fare con dati non formattati. Le due funzioni che si usano per l'I/O binario sono `fread` ed `fwrite`; i rispettivi prototipi sono:

<pre>#include <stdio.h> size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)</pre>	<p>Legge i dati da uno <i>stream</i> ad un buffer.</p> <p>Scrive i dati da un buffer su uno <i>stream</i>.</p>
--	--

<p>Le funzioni ritornano il numero di elementi letti o scritti, in caso di errore o fine del file viene restituito un numero di elementi inferiore al richiesto.</p>
--

Le funzioni rispettivamente leggono e scrivono `nmemb` elementi di dimensione `size` dal buffer `ptr` al file `stream`. In genere si usano queste funzioni quando si devono trasferire su file blocchi di dati binari in maniera compatta e veloce; un primo caso di uso tipico è quello in cui si salva un vettore (o un certo numero dei suoi elementi) con una chiamata del tipo:

```
int WriteVect(FILE *stream, double *vec, size_t nelelem)
{
    int size, nread;
    size = sizeof(*vec);
    if ( (nread = fwrite(vec, size, nelelem, stream)) != nelelem) {
        perror("Write_error");
    }
    return nread;
}
```

in questo caso devono essere specificate le dimensioni di ciascun elemento ed il numero di quelli che si vogliono scrivere. Un secondo caso è invece quello in cui si vuole trasferire su file una struttura; si avrà allora una chiamata tipo:

```
struct histogram {
    int nbins;
    double max, min;
    double *bin;
} histo;

int WriteStruct(FILE *stream, struct histogram *histo)
{
    if ( fwrite(histo, sizeof(*histo), 1, stream) !=1) {
        perror("Write_error");
    }
    return nread;
}
```

in cui si specifica la dimensione dell'intera struttura ed un solo elemento.

In realtà quello che conta nel trasferimento dei dati sono le dimensioni totali, che sono sempre pari al prodotto `size * nelelem`, la differenza sta nel fatto che le funzioni non ritornano il numero di byte scritti, ma il numero di elementi (e con questo possono facilitare i conti).

La funzione `fread` legge sempre un numero intero di elementi, se incontra la fine del file l'oggetto letto parzialmente viene scartato (lo stesso avviene in caso di errore). In questo caso la posizione dello *stream* viene impostata alla fine del file (e non a quella corrispondente alla quantità di dati letti).

In caso di errore (o fine del file per `fread`) entrambe le funzioni restituiscono il numero di oggetti effettivamente letti o scritti, che sarà inferiore a quello richiesto. Contrariamente a quanto avviene per i file descriptor, questo segnala una condizione di errore e occorrerà usare `feof` e `ferror` per stabilire la natura del problema.

Benché queste funzioni assicurino la massima efficienza per il salvataggio dei dati, i dati memorizzati attraverso di esse presentano lo svantaggio di dipendere strettamente dalla piattaforma di sviluppo usata ed in genere possono essere riletti senza problemi solo dallo stesso programma che li ha prodotti.

Infatti diversi compilatori possono eseguire ottimizzazioni diverse delle strutture dati e alcuni compilatori (come il `gcc`) possono anche scegliere se ottimizzare l'occupazione di spazio, impacchettando più strettamente i dati, o la velocità inserendo opportuni *padding* per l'allineamento dei medesimi generando quindi output binari diversi. Inoltre altre incompatibilità si possono presentare quando entrano in gioco differenze di architettura hardware, come la dimensione del bus o la modalità di ordinamento dei bit o il formato delle variabili in floating point.

Per questo motivo quando si usa l'input/output binario occorre sempre prendere le opportune precauzioni come usare un formato di più alto livello che permetta di recuperare l'informazione completa, per assicurarsi che versioni diverse del programma siano in grado di rileggere i dati, tenendo conto delle eventuali differenze.

La seconda modalità di input/output non formattato è quella a caratteri, in cui si trasferisce un carattere alla volta. Le funzioni per la lettura a caratteri sono tre, `fgetc`, `getc` e `getchar`, ed i rispettivi prototipi sono:

```
#include <stdio.h>
int  getc(FILE *stream)
int  fgetc(FILE *stream)

int  getchar(void)
```

Leggono un singolo byte da uno *stream*.

Legge un byte dallo *standard input*.

Le funzioni ritornano il byte letto in caso di successo e EOF per un errore o se si arriva alla fine del file.

La funzione `getc` legge un byte da *stream* e lo restituisce come intero, ed in genere è implementata come una macro per cui può avere *side effects*, mentre `fgetc` è assicurato essere sempre una funzione. Infine `getchar` è equivalente a `getc(stdin)`.

A parte `getchar`, che si usa in genere per leggere un carattere da tastiera, le altre due funzioni sono sostanzialmente equivalenti. La differenza è che `getc` è ottimizzata al massimo e normalmente viene implementata con una macro, per cui occorre stare attenti a cosa le si passa come argomento, infatti *stream* può essere valutato più volte nell'esecuzione, e non viene passato in copia con il meccanismo visto in sez. 2.4.1; per questo motivo se si passa un'espressione si possono avere effetti indesiderati.

Invece `fgetc` è assicurata essere sempre una funzione, per questo motivo la sua esecuzione normalmente è più lenta per via dell'overhead della chiamata, ma è altresì possibile ricavarne l'indirizzo, che può essere passato come argomento ad un'altra funzione (e non si hanno i problemi accennati in precedenza nel tipo di argomento).

Le tre funzioni restituiscono tutte un `unsigned char` convertito ad `int` (si usa `unsigned char` in modo da evitare l'espansione del segno). In questo modo il valore di ritorno è sempre positivo, tranne in caso di errore o fine del file.

Nelle estensioni GNU che provvedono la localizzazione sono definite tre funzioni equivalenti alle precedenti, `getwc`, `fgetwc` e `getwchar`, che invece di un carattere di un byte restituiscono un carattere in formato esteso (cioè di tipo `wint_t`), il loro prototipo è:

<pre>#include <stdio.h> #include <wchar.h> wint_t getwc(FILE *stream) wint_t fgetwc(FILE *stream)</pre>	Leggono un carattere da uno <i>stream</i> .
<pre>wint_t getwchar(void)</pre>	Legge un carattere dallo <i>standard input</i> .
Le funzioni ritornano il carattere letto in caso di successo e WEOF per un errore o se si arriva alla fine del file.	

La funzione `getwc` legge un carattere esteso da `stream` e lo restituisce come intero, ed in genere è implementata come una macro, mentre `fgetwc` è assicurata essere sempre una funzione. Infine `getwchar` è equivalente a `getwc(stdin)`.

Per scrivere un carattere si possono usare tre funzioni, analoghe alle precedenti usate per leggere: `putc`, `fputc` e `putchar`; i loro prototipi sono:

<pre>#include <stdio.h> int putc(int c, FILE *stream) int fputc(int c, FILE *stream)</pre>	Scrivono un byte su uno <i>stream</i> .
<pre>int putchar(int c)</pre>	Scrivono un byte sullo <i>standard output</i> .
Le funzioni ritornano il valore del byte scritto in caso di successo e EOF per un errore.	

La funzione `putc` scrive un byte su `stream` e lo restituisce come intero, ed in genere è implementata come una macro, mentre `fputc` è assicurata essere sempre una funzione. Infine `putchar` è equivalente a `putc(stdout)`. Tutte queste funzioni scrivono sempre un byte alla volta, anche se prendono come argomento un `int` (che pertanto deve essere ottenuto con un cast da un `unsigned char`). Anche il valore di ritorno è sempre un intero; in caso di errore o fine del file il valore di ritorno è EOF.

Come nel caso dell'I/O binario con `fread` e `fwrite` la *glibc* provvede come estensione, per ciascuna delle funzioni precedenti, un'ulteriore funzione, il cui nome è ottenuto aggiungendo un `_unlocked`, che esegue esattamente le stesse operazioni, evitando però il lock implicito dello *stream*.

Per compatibilità con SVID sono inoltre provviste anche due funzioni, `getw` e `putw`, da usare per leggere e scrivere una *word* (cioè due byte in una volta); i loro prototipi sono:

<pre>#include <stdio.h> getw(FILE *stream)</pre>	Legge una parola da uno <i>stream</i> .
<pre>int putw(int w, FILE *stream)</pre>	Scrivono una parola su uno <i>stream</i> .
Le funzioni ritornano la parola letta o scritta in caso di successo e EOF per un errore.	

Le funzioni leggono e scrivono una *word* di due byte, usando comunque una variabile di tipo `int`; il loro uso è deprecato in favore dell'uso di `fread` e `fwrite`, in quanto non è possibile distinguere il valore -1 da una condizione di errore che restituisce EOF.

Uno degli usi più frequenti dell'input/output a caratteri è nei programmi di *parsing* in cui si analizza il testo; in questo contesto diventa utile poter analizzare il carattere successivo da uno *stream* senza estrarlo effettivamente (la tecnica è detta *peeking ahead*) in modo che il programma possa regolarsi avendo dato una *sbirciatina* a quello che viene dopo.

Nel nostro caso questo tipo di comportamento può essere realizzato prima leggendo il carattere, e poi rimandandolo indietro, cosicché ridiventi disponibile per una lettura successiva; la funzione che inverte la lettura si chiama `ungetc` ed il suo prototipo è:

```
#include <stdio.h>
int ungetc(int c, FILE *stream)
```

Manda indietro un byte su uno *stream*.

La funzione ritorna il byte inviato in caso di successo e EOF per un errore.

La funzione rimanda indietro il carattere `c`, con un cast a `unsigned char`, sullo *stream* `stream`. Benché lo standard ANSI C preveda che l'operazione possa essere ripetuta per un numero arbitrario di caratteri, alle implementazioni è richiesto di garantire solo un livello; questo è quello che fa la *glibc*, che richiede che avvenga un'altra operazione fra due `ungetc` successive.

Non è necessario che il carattere che si manda indietro sia l'ultimo che si è letto, e non è necessario neanche avere letto nessun carattere prima di usare `ungetc`, ma di norma la funzione è intesa per essere usata per rimandare indietro l'ultimo carattere letto. Nel caso `c` sia un EOF la funzione non fa nulla, e restituisce sempre EOF; così si può usare `ungetc` anche con il risultato di una lettura alla fine del file.

Se si è alla fine del file si può comunque rimandare indietro un carattere, il flag di *end-of-file* verrà automaticamente cancellato perché c'è un nuovo carattere disponibile che potrà essere riletto successivamente.

Infine si tenga presente che `ungetc` non altera il contenuto del file, ma opera esclusivamente sul buffer interno. Se si esegue una qualunque delle operazioni di riposizionamento (vedi sez. 5.3.4) i caratteri rimandati indietro vengono scartati.

La terza ed ultima modalità di input/output non formattato è quella di linea, in cui si legge o si scrive una riga alla volta. Questa è la modalità usata normalmente per l'I/O da terminale, ed è anche quella che presenta le caratteristiche più controverse.

Le funzioni previste dallo standard ANSI C per leggere una linea sono sostanzialmente due, `gets` e `fgets`, i cui rispettivi prototipi sono:

```
#include <stdio.h>
char *gets(char *string)
```

Legge una linea di testo dallo *standard input*.

```
char *fgets(char *string, int size, FILE *stream)
```

Legge una linea di testo da uno *stream*.

Le funzioni ritornano l'indirizzo della stringa con la linea di testo letta o scritta in caso di successo e NULL per un errore.

Entrambe le funzioni effettuano la lettura, dal file specificato `fgets`, dallo *standard input* `gets`, di una linea di caratteri terminata dal carattere ASCII di *newline*, che come detto corrisponde a quello generato dalla pressione del tasto di invio sulla tastiera. Si tratta del carattere che indica la terminazione di una riga (in sostanza del carattere di “a capo”) che viene rappresentato nelle stringhe di formattazione che vedremo in sez. 5.3.6 come “\n”. Nell'esecuzione delle funzioni `gets` sostituisce “\n” con uno zero, mentre `fgets` aggiunge uno zero dopo il *newline*, che resta dentro la stringa.

Se la lettura incontra la fine del file (o c'è un errore) viene restituito un puntatore NULL, ed il buffer `buf` non viene toccato. L'uso di `gets` è deprecato e deve essere assolutamente evitato,

la funzione infatti non controlla il numero di byte letti, per cui nel caso la stringa letta superi le dimensioni del buffer, si avrà un *buffer overflow*, con sovrascrittura della memoria del processo adiacente al buffer.⁴¹

Questa è una delle vulnerabilità più sfruttate per guadagnare accessi non autorizzati al sistema (i cosiddetti *exploit*), basta infatti inviare una stringa sufficientemente lunga ed opportunamente forgiata per sovrascrivere gli indirizzi di ritorno nello *stack* (supposto che la *gets* sia stata chiamata da una subroutine), in modo da far ripartire l'esecuzione nel codice inviato nella stringa stessa, che in genere contiene uno *shell code*, cioè una sezione di programma che lancia una shell da cui si potranno poi eseguire altri programmi.

La funzione *fgets* non ha i precedenti problemi di *gets* in quanto prende in ingresso la dimensione del buffer *size*, che non verrà mai ecceduta in lettura. La funzione legge fino ad un massimo di *size* caratteri (*newline* compreso), ed aggiunge uno zero di terminazione; questo comporta che la stringa possa essere al massimo di *size-1* caratteri. Se la linea eccede la dimensione del buffer verranno letti solo *size-1* caratteri, ma la stringa sarà sempre terminata correttamente con uno zero finale; sarà possibile leggere i rimanenti caratteri in una chiamata successiva.

Per la scrittura di una linea lo standard ANSI C prevede altre due funzioni, *fputs* e *puts*, analoghe a quelle di lettura, i rispettivi prototipi sono:

```
#include <stdio.h>
int puts(char *string)
int fputs(char *string, int size, FILE *stream)
```

Scrivere una linea di testo sullo *standard output*.
Scrivere una linea di testo su uno *stream*.

Le funzioni ritornano un valore non negativo in caso di successo e EOF per un errore.

La funzione *puts* scrive una linea di testo mantenuta all'indirizzo *string* sullo *standard output* mentre *puts* la scrive sul file indicato da *stream*. Dato che in questo caso si scrivono i dati in uscita *puts* non ha i problemi di *gets* ed è in genere la forma più immediata per scrivere messaggi sullo *standard output*; la funzione prende una stringa terminata da uno zero ed aggiunge automaticamente il ritorno a capo. La differenza con *fputs* (a parte la possibilità di specificare un file diverso da *stdout*) è che quest'ultima non aggiunge il *newline*, che deve essere previsto esplicitamente.

Come per le analoghe funzioni di input/output a caratteri, anche per l'I/O di linea esistono delle estensioni per leggere e scrivere linee di caratteri estesi, le funzioni in questione sono *fgetws* e *fputws* ed i loro prototipi sono:

```
#include <wchar.h>
wchar_t *fgetws(wchar_t *ws, int n, FILE *stream)
int fputws(const wchar_t *ws, FILE *stream)
```

Legge una stringa di carattere estesi da uno *stream*.
Scrive una stringa di carattere estesi da uno *stream*.

Le funzioni ritornano rispettivamente l'indirizzo della stringa o un non negativo in caso di successo e NULL o EOF per un errore o per la fine del file.

La funzione *fgetws* legge un massimo di *n* caratteri estesi dal file *stream* al buffer *ws*, mentre la funzione *fputws* scrive la linea *ws* di caratteri estesi sul file indicato da *stream*. Il comportamento di queste due funzioni è identico a quello di *fgets* e *fputs*, a parte il fatto che tutto (numero di caratteri massimo, terminatore della stringa, *newline*) è espresso in termini di caratteri estesi anziché di normali caratteri ASCII.

⁴¹questa tecnica è spiegata in dettaglio e con molta efficacia nell'ormai famoso articolo di Aleph1 [?].

Come abbiamo visto, le funzioni di lettura per l'input/output di linea previste dallo standard ANSI C presentano svariati inconvenienti. Benché `fgets` non abbia i gravissimi problemi di `gets`, può comunque dare risultati ambigui se l'input contiene degli zeri; questi infatti saranno scritti sul buffer di uscita e la stringa in output apparirà come più corta dei byte effettivamente letti. Questa è una condizione che è sempre possibile controllare (deve essere presente un *newline* prima della effettiva conclusione della stringa presente nel buffer), ma a costo di una complicazione ulteriore della logica del programma. Lo stesso dicasi quando si deve gestire il caso di stringa che eccede le dimensioni del buffer.

Per questo motivo la *glibc* prevede, come estensione GNU, due nuove funzioni per la gestione dell'input/output di linea, il cui uso permette di risolvere questi problemi. L'uso di queste funzioni deve essere attivato definendo la macro `_GNU_SOURCE` prima di includere `stdio.h`. La prima delle due, `getline`, serve per leggere una linea terminata da un *newline*, esattamente allo stesso modo di `fgets`, il suo prototipo è:

<pre>#include <stdio.h> ssize_t getline(char **buffer, size_t *n, FILE *stream)</pre>	Legge una riga da uno <i>stream</i> .
La funzione ritorna il numero di caratteri letti in caso di successo e <code>-1</code> per un errore o per il raggiungimento della fine del file.	

La funzione legge una linea dal file `stream` copiandola sul buffer indicato da `buffer` riallocandola se necessario (l'indirizzo del buffer e la sua dimensione vengono sempre riscritte). Permette così di eseguire una lettura senza doversi preoccupare della eventuale lunghezza eccessiva della stringa da leggere.

Essa prende come primo argomento l'indirizzo del puntatore al buffer su cui si vuole copiare la linea. Quest'ultimo *deve* essere stato allocato in precedenza con una `malloc`: non si può cioè passare come argomento primo argomento l'indirizzo di un puntatore ad una variabile locale. Come secondo argomento la funzione vuole l'indirizzo della variabile contenente le dimensioni del suddetto buffer.

Se il buffer di destinazione è sufficientemente ampio la stringa viene scritta subito, altrimenti il buffer viene allargato usando `realloc` e la nuova dimensione ed il nuovo puntatore vengono restituiti indietro, si noti infatti come entrambi gli argomenti siano dei *value result argument*, per i quali vengono passati dei puntatori anziché i valori delle variabili, secondo quanto abbiamo descritto in sez. 2.4.1).

Se si passa alla funzione l'indirizzo di un puntatore impostato a `NULL` e `*n` è zero, la funzione provvede da sola all'allocazione della memoria necessaria a contenere la linea. In tutti i casi si ottiene dalla funzione un puntatore all'inizio del testo della linea letta. Un esempio di codice può essere il seguente:

```
size_t n = 0;
char *ptr = NULL;
int nread;
FILE * file;
...
nread = getline(&ptr, &n, file);
```

e per evitare *memory leak* occorre ricordarsi di liberare la memoria allocata dalla funzione eseguendo una `free` su `ptr`.

Il valore di ritorno di `getline` indica il numero di caratteri letti dallo *stream*, quindi compreso il *newline*, ma non lo zero di terminazione. Questo permette anche di distinguere anche gli eventuali zeri letti come dati dallo *stream* da quello inserito dalla funzione dopo il

newline per terminare la stringa. Se si è alla fine del file e non si è potuto leggere nulla o se c'è stato un errore la funzione restituisce `-1`.

La seconda estensione GNU per la lettura con l'I/O di linea è una generalizzazione di `getline` per poter usare come separatore delle linee un carattere qualsiasi al posto del *newline*. La funzione si chiama `getdelim` ed il suo prototipo è:

```
#include <stdio.h>
size_t getdelim(char **buffer, size_t *n, int delim, FILE *stream)
    Legge da uno stream una riga delimitata da un carattere scelto.
```

La funzione ha gli stessi valori di ritorno e gli stessi errori di `getline`.

La funzione è identica a `getline` solo che usa `delim` al posto del carattere di *newline* come separatore di linea. Il comportamento di `getdelim` è identico a quello di `getline`, che può essere implementata da `getdelim` passando `"\n"` come valore dell'argomento `delim`.

5.3.6 Input/output formattato

L'ultima modalità di input/output è quella formattata, che è una delle caratteristiche più utilizzate delle librerie standard del C; in genere questa è la modalità in cui si esegue normalmente l'output su terminale poiché permette di stampare in maniera facile e veloce dati, tabelle e messaggi.

L'output formattato viene eseguito con una delle 13 funzioni della famiglia `printf`; le tre più usate sono `printf`, `fprintf` e `sprintf`, i cui prototipi sono:

```
#include <stdio.h>
int printf(const char *format, ...)           Scrive una stringa formattata sullo standard output.
int fprintf(FILE *stream, const char *format, ...) Scrive una stringa formattata su uno stream.
int sprintf(char *str, const char *format, ...) Scrive una stringa formattata su un buffer.
```

Le funzioni ritornano il numero di caratteri scritti in caso di successo e un valore negativo per un errore.

Le funzioni usano la stringa `format` come indicatore del formato con cui dovrà essere scritto il contenuto degli argomenti, il cui numero è variabile e dipende dal formato stesso.

Le prime due servono per scrivere su file (lo *standard output* o quello specificato) la terza permette di scrivere su una stringa, in genere l'uso di `sprintf` è sconsigliato in quanto è possibile, se non si ha la sicurezza assoluta sulle dimensioni del risultato della stampa, eccedere le dimensioni di `str`, con conseguente sovrascrittura di altre variabili e possibili *buffer overflow*. Per questo motivo si consiglia l'uso dell'alternativa `snprintf`, il cui prototipo è:

```
#include <stdio.h>
snprintf(char *str, size_t size, const char *format, ...)
    Scrive una stringa formattata su un buffer.
```

La funzione ha lo stesso valore di ritorno e gli stessi errori di `sprintf`.

la funzione è identica a `sprintf`, ma non scrive su `str` più di `size` caratteri, garantendo così che il buffer non possa essere sovrascritto.

La parte più complessa delle funzioni di scrittura formattata è il formato della stringa `format` che indica le conversioni da fare, e da cui deriva anche il numero degli argomenti che dovranno essere passati a seguire: si noti come tutte queste funzioni siano *"variadic"*,

Valore	Tipo	Significato
%d	int	Stampa un numero intero in formato decimale con segno.
%i	int	Identico a %d in output.
%o	unsigned int	Stampa un numero intero come ottale.
%u	unsigned int	Stampa un numero intero in formato decimale senza segno.
%x, %X	unsigned int	Stampano un intero in formato esadecimale, rispettivamente con lettere minuscole e maiuscole.
%f	double	Stampa un numero in virgola mobile con la notazione a virgola fissa.
%e, %E	double	Stampano un numero in virgola mobile con la notazione esponenziale, rispettivamente con lettere minuscole e maiuscole.
%g, %G	double	Stampano un numero in virgola mobile con la notazione più appropriate delle due precedenti, rispettivamente con lettere minuscole e maiuscole.
%a, %A	double	Stampano un numero in virgola mobile in notazione esadecimale frazionaria.
%c	int	Stampa un carattere singolo.
%s	char *	Stampa una stringa.
%p	void *	Stampa il valore di un puntatore.
%n	&int	Prende il numero di caratteri stampati finora.
%%		Stampa un "%".

Tabella 5.13: Valori possibili per gli specificatori di conversione in una stringa di formato di printf.

prendendo un numero di argomenti variabile che dipende appunto da quello che si è specificato in `format`.

La stringa di formato è costituita da caratteri normali (tutti eccetto "%"), che vengono passati invariati in uscita, e da direttive di conversione, in cui devono essere sempre presenti il carattere "%", che introduce la direttiva, ed uno degli specificatori di conversione (riportati in tab. 5.13) che la conclude.

Il formato di una direttiva di conversione prevede una serie di possibili elementi opzionali oltre al carattere "%" e allo specificatore di conversione. In generale essa è sempre del tipo:

% [n. parametro \$] [flag] [[larghezza] [. precisione]] [tipo] conversione

in cui tutti i valori tranne il "%" e lo specificatore di conversione sono opzionali (e per questo sono indicati fra parentesi quadre); si possono usare più elementi opzionali, nel qual caso devono essere specificati in questo ordine:

- uno specificatore del parametro da usare (terminato da un carattere "\$"),
- uno o più flag (i cui valori possibili sono riassunti in tab. 5.14) che controllano il formato di stampa della conversione,
- uno specificatore di larghezza (un numero decimale), eventualmente seguito (per i numeri in virgola mobile) da un specificatore di precisione (un altro numero decimale),
- uno specificatore del tipo di dato, che ne indica la dimensione (i cui valori possibili sono riassunti in tab. 5.15).

Valore	Significato
#	Chiede la conversione in forma alternativa.
0	La conversione è riempita con zeri alla sinistra del valore.
-	La conversione viene allineata a sinistra sul bordo del campo.
' '	Mette uno spazio prima di un numero con segno di valore positivo.
+	Mette sempre il segno (+ o -) prima di un numero.

Tabella 5.14: I valori dei flag per il formato di printf

Dettagli ulteriori sulle varie opzioni di stampa e su tutte le casistiche dei vari formati possono essere trovati nella pagina di manuale di `printf` e nella documentazione della *glibc*.

Valore	Significato
hh	Una conversione intera corrisponde a un char con o senza segno, o il puntatore per il numero dei parametri n è di tipo char.
h	Una conversione intera corrisponde a uno short con o senza segno, o il puntatore per il numero dei parametri n è di tipo short.
l	Una conversione intera corrisponde a un long con o senza segno, o il puntatore per il numero dei parametri n è di tipo long, o il carattere o la stringa seguenti sono in formato esteso.
ll	Una conversione intera corrisponde a un long long con o senza segno, o il puntatore per il numero dei parametri n è di tipo long long.
L	Una conversione in virgola mobile corrisponde a un double.
q	Sinonimo di ll.
j	Una conversione intera corrisponde a un intmax_t o uintmax_t.
z	Una conversione intera corrisponde a un size_t o ssize_t.
t	Una conversione intera corrisponde a un ptrdiff_t.

Tabella 5.15: Il modificatore di tipo di dato per il formato di printf

Una versione alternativa delle funzioni di output formattato, che permettono di usare il puntatore ad una lista variabile di argomenti (vedi sez. 2.4.2), sono `vprintf`, `vfprintf` e `vsprintf`, i cui prototipi sono:

```
#include <stdio.h>
int vprintf(const char *format, va_list ap)
    Scrive una stringa formattata sullo standard output.
int fprintf(FILE *stream, const char *format, va_list ap)
    Scrive una stringa formattata su uno stream.
int vsprintf(char *str, const char *format, va_list ap)
    Scrive una stringa formattata su un buffer.

Le funzioni ritornano il numero di caratteri scritti in caso di successo e un valore negativo
per un errore.
```

Con queste funzioni è possibile selezionare gli argomenti da passare ad una funzione di stampa indicando direttamente la lista tramite l'argomento `ap`. Per poter far questo ovviamente la lista variabile degli argomenti dovrà essere trattata come visto in sez. 2.4.2, e dopo l'esecuzione della funzione l'argomento `ap` non sarà più utilizzabile (in generale dovrebbe essere eseguito un `va_end(ap)` ma in Linux questo non è necessario).

Come per `sprintf` anche per `vsprintf` esiste una analoga `vsnprintf` che pone un limite sul numero di caratteri che vengono scritti sulla stringa di destinazione:

```
#include <stdio.h>
vsprintf(char *str, size_t size, const char *format, va_list ap)
    Scrive una stringa formattata su un buffer.

La funzione ha lo stesso valore di ritorno e gli stessi errori di vsprintf.
```

in modo da evitare possibili *buffer overflow*.

Per eliminare alla radice questi problemi, la *glibc* supporta una specifica estensione GNU che alloca dinamicamente tutto lo spazio necessario; l'estensione si attiva al solito definendo `_GNU_SOURCE`, le due funzioni sono `asprintf` e `vasprintf`, ed i rispettivi prototipi sono:

```
#include <stdio.h>
int asprintf(char **strptr, const char *format, ...)
int vasprintf(char **strptr, const char *format, va_list ap)
    Scrive una stringa formattata su un buffer.

Le funzioni hanno lo stesso valore di ritorno e gli stessi errori di vsprintf.
```


Entrambe le funzioni prendono come argomento `strptr` che deve essere l'indirizzo di un puntatore ad una stringa di caratteri, in cui verrà restituito (si ricordi quanto detto in sez. 2.4.1 a proposito dei *value result argument*) l'indirizzo della stringa allocata automaticamente dalle funzioni. Occorre inoltre ricordarsi di invocare `free` per liberare detto puntatore quando la stringa non serve più, onde evitare *memory leak*.

Infine una ulteriore estensione GNU definisce le due funzioni `dprintf` e `vdprintf`, che prendono un file descriptor al posto dello *stream*. Altre estensioni permettono di scrivere con caratteri estesi. Anche queste funzioni, il cui nome è generato dalle precedenti funzioni aggiungendo una *w* davanti a `print`, sono trattate in dettaglio nella documentazione della *glibc*.

In corrispondenza alla famiglia di funzioni `printf` che si usano per l'output formattato, l'input formattato viene eseguito con le funzioni della famiglia `scanf`; fra queste le tre più importanti sono `scanf`, `fscanf` e `sscanf`, i cui prototipi sono:

```
#include <stdio.h>
int scanf(const char *format, ...)           Esegue la scansione di dati dallo standard input.
int fscanf(FILE *stream, const char *format, ...) Esegue la scansione di dati da uno stream.
int sscanf(char *str, const char *format, ...) Esegue la scansione di dati da un buffer.
```

La funzione ritorna il numero di elementi assegnati in caso di successo e EOF per un errore o se si raggiunta la fine del file.

Le funzioni eseguono una scansione della rispettiva fonte di input cercando una corrispondenza di quanto letto con il formato dei dati specificato da `format`, ed effettuano le relative conversioni memorizzando il risultato negli argomenti seguenti, il cui numero è variabile e dipende dal valore di `format`. Come per le analoghe funzioni di scrittura esistono le relative `vscanf`, `vfscanf` e `vsscanf` che usano un puntatore ad una lista di argomenti. Le funzioni ritornano il numero di elementi assegnati. Questi possono essere in numero inferiore a quelli specificati, ed anche zero. Quest'ultimo valore significa che non si è trovata corrispondenza.

Tutte le funzioni della famiglia delle `scanf` vogliono come argomenti i puntatori alle variabili che dovranno contenere le conversioni; questo è un primo elemento di disagio in quanto è molto facile dimenticarsi di questa caratteristica.

Le funzioni leggono i caratteri dallo *stream* (o dalla stringa) di input ed eseguono un confronto con quanto indicato in `format`, la sintassi di questo argomento è simile a quella usata per l'analogo di `printf`, ma ci sono varie differenze. Le funzioni di input infatti sono più orientate verso la lettura di testo libero che verso un input formattato in campi fissi. Uno spazio in `format` corrisponde con un numero qualunque di caratteri di separazione (che possono essere spazi, tabulatori, virgole ecc.), mentre caratteri diversi richiedono una corrispondenza esatta. Le direttive di conversione sono analoghe a quelle di `printf` e si trovano descritte in dettaglio nelle pagine di manuale e nel manuale della *glibc*.

Le funzioni eseguono la lettura dall'input, scartano i separatori (e gli eventuali caratteri diversi indicati dalla stringa di formato) effettuando le conversioni richieste; in caso la corrispondenza fallisca (o la funzione non sia in grado di effettuare una delle conversioni richieste) la scansione viene interrotta immediatamente e la funzione ritorna lasciando posizionato lo *stream* al primo carattere che non corrisponde.

Data la notevole complessità di uso di queste funzioni, che richiedono molta cura nella definizione delle corrette stringhe di formato e sono facilmente soggette ad errori, e considerato anche il fatto che è estremamente macchinoso recuperare in caso di fallimento nelle corrispon-

denze, l'input formattato non è molto usato. In genere infatti quando si ha a che fare con un input relativamente semplice si preferisce usare l'input di linea ed effettuare scansione e conversione di quanto serve direttamente con una delle funzioni di conversione delle stringhe; se invece il formato è più complesso diventa più facile utilizzare uno strumento come *flex*⁴² per generare un analizzatore lessicale o *bison*⁴³ per generare un parser.

5.4 Funzioni avanzate

In questa sezione esamineremo alcune funzioni avanzate che permettono di eseguire operazioni di basso livello nella gestione degli *stream*, come leggerne gli attributi, controllarne le modalità di bufferizzazione, gestire in maniera esplicita i lock impliciti presenti ad uso della programmazione *multi-thread*.

5.4.1 Le funzioni di controllo

Al contrario di quanto avviene con i file descriptor, le librerie standard del C non prevedono nessuna funzione come la *fcntl* per il controllo degli attributi dei file. Però, dato che ogni *stream* si appoggia ad un file descriptor, si può usare la funzione *fileno* per ottenere il valore di quest'ultimo; il suo prototipo è:

```
#include <stdio.h>
int fileno(FILE *stream)
```

Legge il file descriptor sottostante lo *stream*.

La funzione ritorna il numero del file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà il valore `EBADF` se *stream* non è valido.

In questo modo diventa possibile usare direttamente *fcntl* sul file descriptor sottostante, ma anche se questo permette di accedere agli attributi del file descriptor sottostante lo *stream*, non ci dà nessuna informazione riguardo alle proprietà dello *stream* medesimo. La *glibc* però supporta alcune estensioni derivate da Solaris, che permettono di ottenere informazioni utili relative allo *stream*.

Ad esempio in certi casi può essere necessario sapere se un certo *stream* è accessibile in lettura o scrittura. In genere questa informazione non è disponibile, e ci si deve ricordare come è stato aperto il file. La cosa può essere complessa se le operazioni vengono effettuate in una subroutine, che a questo punto necessiterà di informazioni aggiuntive rispetto al semplice puntatore allo *stream*. Questo problema può essere risolto con le due funzioni `__freadable` e `__fwritable` i cui prototipi sono:

```
#include <stdio_ext.h>
int __freadable(FILE *stream)

int __fwritable(FILE *stream)
```

Controlla se uno *stream* consente la lettura.

Controlla se uno *stream* consente la scrittura.

Le funzioni ritornano un valore diverso da 0 se l'operazione richiesta è consentita, non sono previste condizioni di errore.

che permettono di ottenere questa informazione.

⁴²il programma *flex*, è una implementazione libera di *lex* un generatore di analizzatori lessicali. Per i dettagli si può fare riferimento al manuale [?].

⁴³il programma *bison* è un clone del generatore di parser *yacc*, maggiori dettagli possono essere trovati nel relativo manuale [?].

La conoscenza dell'ultima operazione effettuata su uno *stream* aperto è utile in quanto permette di trarre conclusioni sullo stato del buffer e del suo contenuto. Altre due funzioni, `__freading` e `__fwriting` servono a tale scopo, il loro prototipo è:

```
#include <stdio_ext.h>
int __freading(FILE *stream)
                                     Controlla l'ultima operazione di lettura.

int __fwriting(FILE *stream)
                                     Controlla l'ultima operazione di scrittura.

Le funzioni ritornano un valore diverso da 0 se l'operazione richiesta è consentita, non sono
previste condizioni di errore.
```

La funzione `__freading` restituisce un valore diverso da zero se *stream* è aperto in sola lettura o se l'ultima operazione è stata di lettura mentre `__fwriting` restituisce un valore diverso da zero se *stream* è aperto in sola scrittura o se l'ultima operazione è stata di scrittura.

Le due funzioni permettono di determinare di che tipo è stata l'ultima operazione eseguita su uno *stream* aperto in lettura/scrittura; ovviamente se uno *stream* è aperto in sola lettura (o sola scrittura) la modalità dell'ultima operazione è sempre determinata; l'unica ambiguità è quando non sono state ancora eseguite operazioni, in questo caso le funzioni rispondono come se una operazione ci fosse comunque stata.

5.4.2 Il controllo della bufferizzazione

Come accennato in sez. 5.3.2 le librerie definiscono una serie di funzioni che permettono di controllare il comportamento degli *stream*; se non si è specificato nulla, la modalità di buffering viene decisa autonomamente sulla base del tipo di file sottostante, ed i buffer vengono allocati automaticamente.

Però una volta che si sia aperto lo *stream* (ma prima di aver compiuto operazioni su di esso) è possibile intervenire sulle modalità di buffering; la funzione che permette di controllare la bufferizzazione è `setvbuf`, il cui prototipo è:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
                                     Imposta la bufferizzazione dello stream.

La funzione ritorna 0 in caso di successo e un altro valore qualunque per un errore, nel qual
caso errno assumerà un valore appropriato.
```

La funzione imposta la bufferizzazione dello *stream* *stream* nella modalità indicata da *mode* con uno dei valori di tab. 5.16, usando *buf* come buffer di lunghezza *size* e permette di controllare tutti gli aspetti della bufferizzazione. L'utente può specificare un buffer da usare al posto di quello allocato dal sistema passandone alla funzione l'indirizzo in *buf* e la dimensione in *size*.

Valore	Modalità
<code>_IONBF</code>	<i>unbuffered</i>
<code>_IOLBF</code>	<i>line buffered</i>
<code>_IOFBF</code>	<i>fully buffered</i>

Tabella 5.16: Valori dell'argomento *mode* di `setvbuf` per l'impostazione delle modalità di bufferizzazione.

Ovviamente se si usa un buffer specificato dall'utente questo deve essere stato allocato e rimanere disponibile per tutto il tempo in cui si opera sullo *stream*. In genere conviene allocarlo con `malloc` e deallocarlo dopo la chiusura del file; ma fintanto che il file è usato all'interno di una funzione, può anche essere usata una variabile automatica. In `stdio.h` è

definita la costante `BUFSIZ`, che indica le dimensioni generiche del buffer di uno *stream*, queste vengono usate dalla funzione `setbuf`. Non è detto però che tale dimensione corrisponda sempre al valore ottimale (che può variare a seconda del dispositivo).

Dato che la procedura di allocazione manuale è macchinosa, comporta dei rischi, come delle scritture accidentali sul buffer, e non assicura la scelta delle dimensioni ottimali, è sempre meglio lasciare allocare il buffer alle funzioni di libreria, che sono in grado di farlo in maniera ottimale e trasparente all'utente (in quanto la deallocazione avviene automaticamente). Inoltre siccome alcune implementazioni usano parte del buffer per mantenere delle informazioni di controllo, non è detto che le dimensioni dello stesso coincidano con quelle su cui viene effettuato l'I/O.

Per evitare che `setvbuf` imposti il buffer basta passare un valore `NULL` per `buf` e la funzione ignorerà l'argomento `size` usando il buffer allocato automaticamente dal sistema. Si potrà comunque modificare la modalità di bufferizzazione, passando in `mode` uno degli opportuni valori elencati in tab. 5.16. Qualora si specifichi la modalità non bufferizzata i valori di `buf` e `size` vengono sempre ignorati.

Oltre a `setvbuf` la *glibc* definisce altre tre funzioni per la gestione della bufferizzazione di uno *stream*: `setbuf`, `setbuffer` e `setlinebuf`, i rispettivi prototipi sono:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf)
void setbuffer(FILE *stream, char *buf, size_t size)
void setlinebuf(FILE *stream)
```

Impostano il buffer per uno *stream*.
Porta uno *stream* in modalità *line buffered*.

Le funzioni non ritornano niente e non hanno condizioni di errore.

La funzione `setbuf` disabilita la bufferizzazione se `buf` è `NULL`, altrimenti usa `buf` come buffer di dimensione `BUFSIZ` in modalità *fully buffered*, mentre `setbuffer` disabilita la bufferizzazione se `buf` è `NULL`, altrimenti usa `buf` come buffer di dimensione `size` in modalità *fully buffered*. Tutte queste funzioni sono realizzate con opportune chiamate a `setvbuf` e sono definite solo per compatibilità con le vecchie librerie BSD, pertanto non è il caso di usarle se non per la portabilità su vecchi sistemi.

Infine la *glibc* provvede le funzioni non standard, anche queste originarie di Solaris, `__flbf` e `__fbufsize` che permettono di leggere le proprietà di bufferizzazione di uno *stream*; i cui prototipi sono:

```
size_t __fbufsize(FILE *stream)
int __flbf(FILE *stream)
```

Restituisce le dimensioni del buffer di uno *stream*.
Controlla la modalità di bufferizzazione di uno *stream*.

Le funzioni ritornano rispettivamente la dimensione del buffer o un valore non nullo se lo *stream* è in modalità *line-buffered*, non sono previste condizioni di errore.

Come già accennato, indipendentemente dalla modalità di bufferizzazione scelta, si può forzare lo scarico dei dati sul file con la funzione `fflush`, il cui prototipo è:

```
#include <stdio.h>
int fflush(FILE *stream)
```

Forza la scrittura dei dati bufferizzati di uno *stream*.

La funzione ritorna 0 in caso di successo e EOF per un errore, nel qual caso `errno` assumerà il valore `EBADF` se *stream* non è aperto o non è aperto in scrittura, o ad uno degli errori di `write`.

Se *stream* è NULL lo scarico dei dati è forzato per tutti gli *stream* aperti. Esistono però circostanze, ad esempio quando si vuole essere sicuri che sia stato eseguito tutto l'output su terminale, in cui serve poter effettuare lo scarico dei dati solo per gli *stream* in modalità *line buffered*. Per fare questo la *glibc* supporta una estensione di Solaris, la funzione `_flushlbf`, il cui prototipo è:

```
#include <stdio-ext.h>
void _flushlbf(void)
```

Forza la scrittura dei dati bufferizzati degli *stream* in modalità *line buffered*.

La funzione non ritorna nulla e non presenta condizioni di errore.

Si ricordi comunque che lo scarico dei dati dai buffer effettuato da queste funzioni non comporta la scrittura di questi su disco; se si vuole che il kernel dia effettivamente avvio alle operazioni di scrittura su disco occorre usare `sync` o `fsync` (si veda sez. 5.2.3).

Infine esistono anche circostanze in cui si vuole scartare tutto l'output pendente; per questo si può usare `fpurge`, il cui prototipo è:

```
#include <stdio.h>
int fpurge(FILE *stream)
```

Cancella i buffer di uno *stream*.

La funzione ritorna 0 in caso di successo e EOF per un errore.

La funzione scarta tutti i dati non ancora scritti (se il file è aperto in scrittura), e tutto l'input non ancora letto (se è aperto in lettura), compresi gli eventuali caratteri rimandati indietro con `ungetc`.

5.4.3 Gli *stream* e i *thread*

Gli *stream* possono essere usati in applicazioni *multi-thread* allo stesso modo in cui sono usati nelle applicazioni normali, ma si deve essere consapevoli delle possibili complicazioni anche quando non si usano i *thread*, dato che l'implementazione delle librerie è influenzata pesantemente dalle richieste necessarie per garantirne l'uso con i *thread*.

Lo standard POSIX richiede che le operazioni sui file siano atomiche rispetto ai *thread*, per questo le operazioni sui buffer effettuate dalle funzioni di libreria durante la lettura e la scrittura di uno *stream* devono essere opportunamente protette, in quanto il sistema assicura l'atomicità solo per le *system call*. Questo viene fatto associando ad ogni *stream* un opportuno blocco che deve essere implicitamente acquisito prima dell'esecuzione di qualunque operazione.

Ci sono comunque situazioni in cui questo non basta, come quando un *thread* necessita di compiere più di una operazione sullo *stream* atomicamente. Per questo motivo le librerie provvedono anche le funzioni `flockfile` e `funlockfile` che permettono la gestione esplicita dei blocchi sugli *stream*. Esse sono disponibili definendo `_POSIX_THREAD_SAFE_FUNCTIONS` ed i loro prototipi sono:

```
#include <stdio.h>
void flockfile(FILE *stream)
```

Acquisisce il lock su uno *stream*.

```
void funlockfile(FILE *stream)
```

Rilascia il lock su uno *stream*.

Le funzioni non ritornano nulla e non sono previste condizioni di errore.

La funzione `flockfile` esegue l'acquisizione del lock dello *stream* `stream`, bloccandosi se questo risulta non è disponibile, mentre `funlockfile` rilascia un lock che si è precedentemente acquisito.

Una terza funzione, che serve a provare ad acquisire un lock senza bloccarsi qualora non sia possibile, è `ftrylockfile`, il cui prototipo è:

```
#include <stdio.h>
int ftrylockfile(FILE *stream)
```

Tenta l'acquisizione del lock di uno *stream*.

La funzione ritorna 0 in caso di acquisizione del lock ed un altro valore qualunque altrimenti, non sono previste condizioni di errore.

Con queste funzioni diventa possibile acquisire un blocco ed eseguire tutte le operazioni volute, per poi rilasciarlo. Ma, vista la complessità delle strutture di dati coinvolte, le operazioni di blocco non sono del tutto indolori, e quando il locking dello *stream* non è necessario (come in tutti i programmi che non usano i *thread*), tutta la procedura può comportare dei costi pesanti in termini di prestazioni.

Per questo motivo alle usuali funzioni di I/O non formattato sono associate delle ulteriori versioni, caratterizzate dall'aggiunta del suffisso `_unlocked`, che possono essere usate quando il locking non serve⁴⁴ con prestazioni molto più elevate, dato che spesso queste versioni (come accade per `getc` e `putc`) sono realizzate come macro.

La sostituzione di tutte le funzioni di I/O con le relative versioni `_unlocked` in un programma che non usa i *thread* è però un lavoro abbastanza noioso. Per questo motivo la *glibc* fornisce al programmatore pigro un'altra via, anche questa mutuata da estensioni introdotte in Solaris, da poter utilizzare per disabilitare in blocco il locking degli *stream*: l'uso della funzione `__fsetlocking`, il cui prototipo è:

```
#include <stdio_ext.h>
int __fsetlocking(FILE *stream, int type)
```

Specifica se abilitare il locking su uno *stream*.

La funzione ritorna stato di locking interno dello *stream*, non sono previste condizioni di errore.

La funzione imposta o legge lo stato della modalità in cui le operazioni di I/O su *stream* vengono effettuate rispetto all'acquisizione implicita del locking a seconda del valore specificato con `type`, che può assumere uno dei valori indicati in tab. 5.17.

Valore	Significato
FSETLOCKING_INTERNAL	Lo <i>stream</i> userà da ora in poi il blocco implicito predefinito.
FSETLOCKING_BYCALLER	Al ritorno della funzione sarà l'utente a dover gestire da solo il locking dello <i>stream</i> .
FSETLOCKING_QUERY	Restituisce lo stato corrente della modalità di blocco dello <i>stream</i> .

Tabella 5.17: Valori dell'argomento `type` di `__fsetlocking` per l'impostazione delle modalità di bufferizzazione.

La funzione, se usata con `FSETLOCKING_QUERY`, non modifica la modalità di operazione ma restituisce lo stato di locking interno dello *stream* con uno dei valori `FSETLOCKING_INTERNAL` o `FSETLOCKING_BYCALLER`.

⁴⁴in certi casi dette funzioni possono essere usate, visto che sono molto più efficienti, anche in caso di necessità di locking, una volta che questo sia stato acquisito manualmente.

Per tutte le funzioni che abbiamo trattato in sez. 5.3 che eseguono I/O sugli *stream* esiste una versione “_unlocked”,⁴⁵ ma nello standard POSIX sono previste solo `getc_unlocked`, `getchar_unlocked`, `putc_unlocked` e `putchar_unlocked`, tutte le altre pur essendo state aggiunte come estensioni dalla *glibc*, non sono standard, anche se sono presenti anche su altri sistemi unix; in generale comunque l'uso di queste funzioni è sconsigliato e non le tratteremo esplicitamente.

⁴⁵non ne esistono per funzioni di informazione come `ftell` dato che queste non hanno bisogno di un blocco, l'elenco completo delle funzioni “_unlocked” comunque è disponibile nella pagina di manuale delle stesse, accessibile con `man unlocked_stdio`.

Capitolo 6

La gestione del sistema, del tempo e degli errori

In questo capitolo tratteremo varie interfacce che attengono agli aspetti più generali del sistema, come quelle per la gestione dei parametri e della configurazione dello stesso, quelle per la lettura dei limiti e delle caratteristiche, quelle per il controllo dell'uso delle risorse dei processi, quelle per la gestione ed il controllo dei filesystem, degli utenti, dei tempi e degli errori.

6.1 La gestione di caratteristiche e parametri del sistema

In questa sezione tratteremo le varie modalità con cui un programma può ottenere informazioni riguardo alle capacità del sistema, e, per quelle per cui è possibile, sul come modificarle. Ogni sistema unix-like infatti è contraddistinto da un gran numero di limiti e costanti che lo caratterizzano, e che possono dipendere da fattori molteplici, come l'architettura hardware, l'implementazione del kernel e delle librerie, le opzioni di configurazione. Il kernel inoltre mette a disposizione l'accesso ad alcuni parametri che possono modificarne il comportamento.

La definizione di queste caratteristiche ed il tentativo di fornire dei meccanismi generali che i programmi possono usare per ricavarle è uno degli aspetti più complessi e controversi con cui le diverse standardizzazioni si sono dovute confrontare, con risultati spesso tutt'altro che chiari. Daremo comunque una descrizione dei principali metodi previsti dai vari standard per ricavare sia le caratteristiche specifiche del sistema che quelle della gestione dei file, e prenderemo in esame le modalità con cui è possibile intervenire sui parametri del kernel.

6.1.1 Limiti e caratteristiche del sistema

Quando si devono determinare le caratteristiche generali del sistema ci si trova di fronte a diverse possibilità; alcune di queste infatti possono dipendere dall'architettura dell'hardware (come le dimensioni dei tipi interi), o dal sistema operativo (come la presenza o meno del gruppo degli identificatori *saved*), altre invece possono dipendere dalle opzioni con cui si è costruito il sistema (ad esempio da come si è compilato il kernel), o dalla configurazione del medesimo; per questo motivo in generale sono necessari due tipi diversi di funzionalità:

- la possibilità di determinare limiti ed opzioni al momento della compilazione.
- la possibilità di determinare limiti ed opzioni durante l'esecuzione.

La prima funzionalità si può ottenere includendo gli opportuni file di intestazione che contengono le costanti necessarie definite come macro di preprocessore, per la seconda invece sono ovviamente necessarie delle funzioni. La situazione è complicata dal fatto che ci sono molti casi in cui alcuni di questi limiti sono fissi in un'implementazione mentre possono variare in un'altra: tutto questo crea una ambiguità che non è sempre possibile risolvere in maniera chiara. In generale quello che succede è che quando i limiti del sistema sono fissi essi vengono definiti come macro di preprocessore nel file `limits.h`, se invece possono variare, il loro valore sarà ottenibile tramite la funzione `sysconf` (che esamineremo a breve).

Costante	Valore	Significato
MB_LEN_MAX	16	Massima dimensione di un carattere esteso.
CHAR_BIT	8	Numero di bit di char.
UCHAR_MAX	255	Massimo di unsigned char.
SCHAR_MIN	-128	Minimo di signed char.
SCHAR_MAX	127	Massimo di signed char.
CHAR_MIN	0 o -128	Minimo di char. ¹
CHAR_MAX	127 o 255	Massimo di char. ²
SHRT_MIN	-32768	Minimo di short.
SHRT_MAX	32767	Massimo di short.
USHRT_MAX	65535	Massimo di unsigned short.
INT_MAX	2147483647	Minimo di int.
INT_MIN	-2147483648	Minimo di int.
UINT_MAX	4294967295	Massimo di unsigned int.
LONG_MAX	2147483647	Massimo di long.
LONG_MIN	-2147483648	Minimo di long.
ULONG_MAX	4294967295	Massimo di unsigned long.

Tabella 6.1: Costanti definite in `limits.h` in conformità allo standard ANSI C.

Lo standard ANSI C definisce dei limiti che sono tutti fissi, pertanto questo saranno sempre disponibili al momento della compilazione. Un elenco, ripreso da `limits.h`, è riportato in tab. 6.1. Come si può vedere per la maggior parte questi limiti attengono alle dimensioni dei dati interi, che sono in genere fissati dall'architettura hardware, le analoghe informazioni per i dati in virgola mobile sono definite a parte, ed accessibili includendo `float.h`.

Costante	Valore	Significato
LLONG_MAX	9223372036854775807	Massimo di long long.
LLONG_MIN	-9223372036854775808	Minimo di long long.
ULLONG_MAX	18446744073709551615	Massimo di unsigned long long.

Tabella 6.2: Macro definite in `limits.h` in conformità allo standard ISO C90.

Lo standard prevede anche un'altra costante, `FOPEN_MAX`, che può non essere fissa e che pertanto non è definita in `limits.h`, essa deve essere definita in `stdio.h` ed avere un valore minimo di 8. A questi valori lo standard ISO C90 ne aggiunge altri tre, relativi al tipo `long long` introdotto con il nuovo standard, i relativi valori sono in tab. 6.2.

Ovviamente le dimensioni dei vari tipi di dati sono solo una piccola parte delle caratteristiche del sistema; mancano completamente tutte quelle che dipendono dalla implementazione dello stesso. Queste, per i sistemi unix-like, sono state definite in gran parte dallo standard POSIX.1, che tratta anche i limiti relativi alle caratteristiche dei file che vedremo in sez. 6.1.2.

Purtroppo la sezione dello standard che tratta questi argomenti è una delle meno chiare, tanto che Stevens, in [?], la porta come esempio di “*standardese*”. Lo standard prevede che ci

¹il valore può essere 0 o `SCHAR_MIN` a seconda che il sistema usi caratteri con segno o meno.

²il valore può essere `UCHAR_MAX` o `SCHAR_MAX` a seconda che il sistema usi caratteri con segno o meno.

Costante	Valore	Significato
ARG_MAX	131072	Dimensione massima degli argomenti passati ad una funzione della famiglia <i>exec</i> .
CHILD_MAX	999	Numero massimo di processi contemporanei che un utente può eseguire.
OPEN_MAX	256	Numero massimo di file che un processo può mantenere aperti in contemporanea.
STREAM_MAX	8	Massimo numero di stream aperti per processo in contemporanea.
TZNAME_MAX	6	Dimensione massima del nome di una <i>timezone</i> (vedi sez. 6.4.3)).
NGROUPS_MAX	32	Numero di gruppi supplementari per processo (vedi sez. 3.2.1).
SSIZE_MAX	32767	Valore massimo del tipo <i>ssize_t</i> .

Tabella 6.3: Costanti per i limiti del sistema.

siano 13 macro che descrivono le caratteristiche del sistema: 7 per le caratteristiche generiche, riportate in tab. 6.3, e 6 per le caratteristiche dei file, riportate in tab. 6.7.

Costante	Valore	Significato
_POSIX_ARG_MAX	4096	Dimensione massima degli argomenti passati ad una funzione della famiglia <i>exec</i> .
_POSIX_CHILD_MAX	6	Numero massimo di processi contemporanei che un utente può eseguire.
_POSIX_OPEN_MAX	16	Numero massimo di file che un processo può mantenere aperti in contemporanea.
_POSIX_STREAM_MAX	8	Massimo numero di stream aperti per processo in contemporanea.
_POSIX_TZNAME_MAX	6	Dimensione massima del nome di una <i>timezone</i> (vedi sez. 6.4.4).
_POSIX_RTSIG_MAX	8	Numero massimo di segnali <i>real-time</i> (vedi sez. 7.5.1).
_POSIX_NGROUPS_MAX	0	Numero di gruppi supplementari per processo (vedi sez. 3.2.1).
_POSIX_SSIZE_MAX	32767	Valore massimo del tipo <i>ssize_t</i> .

Tabella 6.4: Macro dei valori minimi di alcune caratteristiche generali del sistema per la conformità allo standard POSIX.1.

Lo standard dice che queste macro devono essere definite in `limits.h` quando i valori a cui fanno riferimento sono fissi, e altrimenti devono essere lasciate indefinite, ed i loro valori dei limiti devono essere accessibili solo attraverso `sysconf`. In realtà queste vengono sempre definite ad un valore generico. Si tenga presente poi che alcuni di questi limiti possono assumere valori molto elevati (come `CHILD_MAX`), e non è pertanto il caso di utilizzarli per allocare staticamente della memoria.

A complicare la faccenda si aggiunge il fatto che POSIX.1 prevede una serie di altre costanti (il cui nome inizia sempre con `_POSIX_`) che definiscono i valori minimi le stesse caratteristiche devono avere, perché una implementazione possa dichiararsi conforme allo standard, alcuni dei questi valori sono riportati in tab. 6.4.

In genere questi valori non servono a molto, la loro unica utilità è quella di indicare un limite superiore che assicura la portabilità senza necessità di ulteriori controlli. Tuttavia molti di essi sono ampiamente superati in tutti i sistemi POSIX in uso oggi. Per questo è sempre meglio utilizzare i valori ottenuti da `sysconf`.

Oltre ai precedenti valori e a quelli relativi ai file elencati in tab. 6.8, che devono essere obbligatoriamente definiti, lo standard POSIX.1 ne prevede molti altri. La lista completa si trova dall'header file `bits/posix1_lim.h`, da non usare mai direttamente (è incluso automaticamente all'interno di `limits.h`). Di questi vale la pena menzionarne alcune macro di uso comune, riportate in tab. 6.5, che non indicano un valore specifico, ma denotano la presenza di alcune funzionalità nel sistema, come il supporto del *job control* o degli identificatori del gruppo *saved*.

Macro	Significato
_POSIX_JOB_CONTROL	Il sistema supporta il <i>job control</i> (vedi sez. 8.1).
_POSIX_SAVED_IDS	Il sistema supporta gli identificatori del gruppo <i>saved</i> (vedi sez. 3.2.1) per il controllo di accesso dei processi.
_POSIX_VERSION	Fornisce la versione dello standard POSIX.1 supportata nel formato YYYYMMML (ad esempio 199009L).

Tabella 6.5: Alcune macro definite in `limits.h` in conformità allo standard POSIX.1.

Oltre allo standard POSIX.1, anche lo standard POSIX.2 definisce una serie di altre costanti. Siccome queste sono principalmente attinenti a limiti relativi alle applicazioni di sistema presenti, come quelli su alcuni parametri delle espressioni regolari o del comando `bc`, non li tratteremo esplicitamente, se ne trova una menzione completa nell'header file `bits/posix2_lim.h`, e alcuni di loro sono descritti nella pagina di manuale di `sysconf` e nel manuale della *glibc*.

Quando uno dei limiti o delle caratteristiche del sistema può variare, per non dover essere costretti a ricompilare un programma tutte le volte che si cambiano le opzioni con cui è compilato il kernel, o alcuni dei parametri modificabili al momento dell'esecuzione, è necessario ottenerne il valore attraverso la funzione `sysconf`, cui prototipo è:

```
#include <unistd.h>
long sysconf(int name)
```

Restituisce il valore di un parametro di sistema.

La funzione ritorna in caso di successo il valore del parametro richiesto, o 1 se si tratta di un'opzione disponibile, 0 se l'opzione non è disponibile e -1 per un errore, nel qual caso però `errno` non viene impostata.

La funzione prende come argomento un intero che specifica quale dei limiti si vuole conoscere. Uno specchietto contenente i principali valori disponibili in Linux è riportato in tab. 6.6, l'elenco completo è contenuto in `bits/confname.h`, ed una lista più esaustiva, con le relative spiegazioni, si può trovare nel manuale della *glibc*.

In generale ogni limite o caratteristica del sistema per cui è definita una macro, sia da ANSI C e ISO C90 che da POSIX.1 e POSIX.2, può essere ottenuto attraverso una chiamata a `sysconf`. Il nome della costante da utilizzare come valore dell'argomento `name` si otterrà aggiungendo `_SC_` ai nomi delle costanti definite dai primi due standard (quelle di tab. 6.3), o sostituendolo a `_POSIX_` per le costanti definite dagli altri due standard (quelle di tab. 6.4).

In linea teorica si dovrebbe fare uso di `sysconf` solo quando la relativa costante di sistema non è definita, quindi con un codice analogo al seguente:

```
get_child_max(void)
{
#ifdef CHILD_MAX
    return CHILD_MAX;
#else
    int val = sysconf(_SC_CHILD_MAX);
    if (val < 0) {
        perror("fatal_error");
        exit(-1);
    }
    return val;
#endif
}
```

Parametro	Macro sostituita	Significato
_SC_ARG_MAX	ARG_MAX	La dimensione massima degli argomenti passati ad una funzione della famiglia <i>exec</i> .
_SC_CHILD_MAX	CHILD_MAX	Il numero massimo di processi contemporanei che un utente può eseguire.
_SC_OPEN_MAX	OPEN_MAX	Il numero massimo di file che un processo può mantenere aperti in contemporanea.
_SC_STREAM_MAX	STREAM_MAX	Il massimo numero di stream che un processo può mantenere aperti in contemporanea. Questo limite è previsto anche dallo standard ANSI C, che specifica la macro <i>FOPEN_MAX</i> .
_SC_TZNAME_MAX	TZNAME_MAX	La dimensione massima di un nome di una <i>timezone</i> (vedi sez. 6.4.4).
_SC_NGROUPS_MAX	NGROUP_MAX	Massimo numero di gruppi supplementari che può avere un processo (vedi sez. 3.2.1).
_SC_SSIZE_MAX	SSIZE_MAX	Valore massimo del tipo di dato <i>ssize_t</i> .
_SC_CLK_TCK	CLK_TCK	Il numero di <i>clock tick</i> al secondo, cioè l'unità di misura del <i>process time</i> (vedi sez. 6.4.1).
_SC_JOB_CONTROL	_POSIX_JOB_CONTROL	Indica se è supportato il <i>job control</i> (vedi sez. 8.1) in stile POSIX.
_SC_SAVED_IDS	_POSIX_SAVED_IDS	Indica se il sistema supporta i <i>saved id</i> (vedi sez. 3.2.1).
_SC_VERSION	_POSIX_VERSION	Indica il mese e l'anno di approvazione della revisione dello standard POSIX.1 a cui il sistema fa riferimento, nel formato YYYYMMML, la revisione più recente è 199009L, che indica il Settembre 1990.

Tabella 6.6: Parametri del sistema leggibili dalla funzione *sysconf*.

ma in realtà con Linux queste costanti sono comunque definite, indicando però un limite generico che non è detto sia corretto; per questo motivo è sempre meglio usare i valori restituiti da *sysconf*.

6.1.2 Limiti e caratteristiche dei file

Come per le caratteristiche generali del sistema anche per i file esistono una serie di limiti (come la lunghezza del nome del file o il numero massimo di link) che dipendono sia dall'implementazione che dal filesystem in uso. Anche in questo caso lo standard prevede alcune macro che ne specificano il valore, riportate in tab. 6.7.

Costante	Valore	Significato
LINK_MAX	8	Numero massimo di link a un file.
NAME_MAX	14	Lunghezza in byte di un nome di file.
PATH_MAX	256	Lunghezza in byte di un <i>pathname</i> .
PIPE_BUF	4096	Byte scrivibili atomicamente in una <i>pipe</i> (vedi sez. 11.1.1).
MAX_CANON	255	Dimensione di una riga di terminale in modo canonico (vedi sez. 8.2.1).
MAX_INPUT	255	Spazio disponibile nella coda di input del terminale (vedi sez. 8.2.1).

Tabella 6.7: Costanti per i limiti sulle caratteristiche dei file.

Come per i limiti di sistema, lo standard POSIX.1 detta una serie di valori minimi anche per queste caratteristiche, che ogni sistema che vuole essere conforme deve rispettare. Le relative macro sono riportate in tab. 6.8 e per esse vale lo stesso discorso fatto per le analoghe di tab. 6.4.

Tutti questi limiti sono definiti in *limits.h*; come nel caso precedente il loro uso è di scarsa utilità in quanto ampiamente superati in tutte le implementazioni moderne. In generale i limiti per i file sono molto più soggetti ad essere variabili rispetto ai limiti generali del sistema; ad

Macro	Valore	Significato
_POSIX_LINK_MAX	8	Numero massimo di link a un file.
_POSIX_NAME_MAX	14	Lunghezza in byte di un nome di file.
_POSIX_PATH_MAX	256	Lunghezza in byte di un <i>pathname</i> .
_POSIX_PIPE_BUF	512	Byte scrivibili atomicamente in una <i>pipe</i> .
_POSIX_MAX_CANON	255	Dimensione di una riga di terminale in modo canonico.
_POSIX_MAX_INPUT	255	Spazio disponibile nella coda di input del terminale.

Tabella 6.8: Costanti dei valori minimi delle caratteristiche dei file per la conformità allo standard POSIX.1.

esempio parametri come la lunghezza del nome del file o il numero di link possono variare da filesystem a filesystem.

Per questo motivo quando si ha a che fare con limiti relativi ai file questi devono essere sempre controllati con la funzione `pathconf`, il cui prototipo è:

```
#include <unistd.h>
long pathconf(char *path, int name)
```

Restituisce il valore di un parametro dei file.

La funzione ritorna il valore del parametro richiesto in caso di successo e `-1` per un errore, nel qual caso `errno` viene impostata ad uno degli errori possibili relativi all'accesso a `path`.

La funzione richiede che si specifichi il limite che si vuole controllare con l'argomento `name`, per il quale si deve usare la relativa costante identificativa, il cui nome si ottiene da quelle descritte in tab. 6.7 e tab. 6.8 con la stessa convenzione già vista con `sysconf`, ma in questo caso con l'uso del suffisso “`_PC_`”.

In questo caso la funzione richiede anche un secondo argomento `path` che specifichi a quale file si fa riferimento, dato che il valore del limite cercato può variare a seconda del filesystem su cui si trova il file. Una seconda versione della funzione, `fpathconf`, opera su un file descriptor invece che su un *pathname*, il suo prototipo è:

```
#include <unistd.h>
long fpathconf(int fd, int name)
```

Restituisce il valore di un parametro dei file.

È identica a `pathconf` solo che utilizza un file descriptor invece di un *pathname*; pertanto gli errori restituiti in `errno` cambiano di conseguenza.

ed il suo comportamento è identico a quello di `pathconf` a parte quello di richiedere l'indicazione di un file descriptor nell'argomento `fd`.

6.1.3 I parametri del kernel ed il filesystem `/proc`

Tradizionalmente la funzione che permette la lettura ed l'impostazione dei parametri del sistema è `sysctl`. Si tratta di una funzione derivata da BSD4.4 ed introdotta su Linux a partire dal kernel 1.3.57, ma oggi il suo uso è totalmente deprecato. Una *system call* `_sysctl` continua ad esistere, ma non dispone più di una interfaccia nella *glibc* ed il suo utilizzo può essere effettuato solo tramite `syscall`, ma di nuovo questo viene sconsigliato in quanto la funzionalità non è più mantenuta e molto probabilmente sarà rimossa nel prossimo futuro.³ Per questo motivo eviteremo di trattarne i particolari.

Lo scopo di `sysctl` era quello di fornire ai programmi una modalità per modificare i parametri di sistema. Questi erano organizzati in maniera gerarchica all'interno di un albero

³a partire dal kernel 2.6.34 la funzione viene inserita nella compilazione del kernel previa esplicita richiesta, ed il suo uso produce avvertimenti nei log del kernel.

e per accedere a ciascuno di essi occorre specificare un percorso attraverso i vari nodi dell'albero, in maniera analoga a come avviene per la risoluzione di un *pathname*.

I parametri accessibili e modificabili attraverso questa funzione sono moltissimi, dipendendo anche dallo stato corrente del kernel, ad esempio dai moduli che sono stati caricati nel sistema. Inoltre non essendo standardizzati i loro nomi possono variare da una versione di kernel all'altra, alcuni esempi di questi parametri sono:

- il nome di dominio,
- i parametri del meccanismo di *paging*,
- il filesystem montato come radice,
- la data di compilazione del kernel,
- i parametri dello stack TCP,
- il numero massimo di file aperti,
- il numero massimo di processi,
- i parametri del *SystemV IPC* (vedi sez. 11.2).

Dato che fin dall'inizio i parametri erano organizzati in una struttura albero, è parso naturale riportare questa organizzazione all'interno del filesystem `/proc`. Questo è un filesystem virtuale il cui contenuto è generato direttamente dal kernel, che non fa riferimento a nessun dispositivo fisico, ma presenta in forma di file e directory i dati di alcune delle strutture interne del kernel. Il suo utilizzo principale, come denuncia il nome stesso, è quello di fornire una interfaccia per ottenere i dati relativi ai processi (venne introdotto a questo scopo su BSD), ma nel corso del tempo il suo uso è stato ampliato.

All'interno di questo filesystem sono pertanto presenti una serie di file che riflettono il contenuto dei parametri del kernel (molti dei quali accessibili in sola lettura) e in altrettante directory, nominate secondo il relativo *PID*, vengono mantenute le informazioni relative a ciascun processo attivo nel sistema (abbiamo già incontrato questa caratteristica in sez. 5.2.4 per accedere ai filedescriptor del processo stesso).

In particolare l'albero dei valori dei parametri di sistema impostabili con `sysctl` viene presentato in forma di una gerarchia di file e directory a partire dalla directory `/proc/sys`, cosicché è possibile accedere al valore di un parametro del kernel tramite il *pathname* ad un file sotto `/proc/sys` semplicemente leggendone il contenuto, così come si può modificare un parametro scrivendo sul file ad esso corrispondente.

Il kernel si occupa di generare al volo il contenuto ed i nomi dei file corrispondenti ai vari parametri che sono presenti, e questo ha il grande vantaggio di rendere accessibili gli stessi ad un qualunque comando di shell e di permettere la navigazione dell'albero in modo da riconoscere quali parametri sono presenti senza dover cercare un valore all'interno di una pagina di manuale.

Inizialmente l'uso del filesystem `/proc` serviva soltanto a replicare l'accesso, con altrettante corrispondenze ai file presenti in `/proc/sys`, ai parametri impostabili tradizionalmente con `sysctl`, ma vista la assoluta naturalità dell'interfaccia, e la sua maggiore efficienza, nelle versioni più recenti del kernel questa è diventata la modalità canonica per modificare i parametri del kernel, evitando di dover ricorrere all'uso di una *system call* specifica, che pur essendo ancora presente prima o poi verrà eliminata.

Nonostante la semplificazione nella gestione ottenuta con l'uso di `/proc/sys` resta il problema generale di conoscere il significato di ciascuno degli innumerevoli parametri che vi si trovano. Purtroppo la documentazione degli stessi spesso risulta incompleta e non aggiornata, ma buona parte di quelli più importanti sono descritti dalla documentazione inclusa nei sorgenti del kernel, nella directory `Documentation/sysctl`.

Ma oltre alle informazioni che sostituiscono quelle ottenibili dalla ormai deprecata `sysctl` dentro `/proc` sono disponibili moltissime altre informazioni, fra cui ad esempio anche quelle fornite dalla funzione di sistema `uname`,⁴ il cui prototipo è:

```
#include <sys/utsname.h>
int uname(struct utsname *info)
```

Restituisce informazioni generali sul sistema.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` può assumere solo il valore `EFAULT`.

La funzione, che viene usata dal comando `uname`, restituisce una serie di informazioni relative al sistema nelle stringhe che costituiscono i campi della struttura `utsname` (la cui definizione è riportata in fig. 6.1) che viene scritta nel buffer puntato dall'argomento `info`.

```
struct utsname {
    char sysname[];      /* Operating system name (e.g., "Linux") */
    char nodename[];     /* Name within "some implementation-defined
                        network" */
    char release[];      /* OS release (e.g., "2.6.28") */
    char version[];      /* OS version */
    char machine[];      /* Hardware identifier */
#ifdef _GNU_SOURCE
    char domainname[];  /* NIS or YP domain name */
#endif
};
```

Figura 6.1: La struttura `utsname`.

Si noti come in fig. 6.1 le dimensioni delle stringhe di `utsname` non sono specificate. Il manuale della *glibc* indica due costanti per queste dimensioni, `_UTSNAME_LENGTH` per i campi standard e `_UTSNAME_DOMAIN_LENGTH` per quello relativo al nome di dominio, altri sistemi usano nomi diversi come `SYS_NMLN` o `_SYS_NMLN` o `UTSLEN` che possono avere valori diversi. Dato che il buffer per `utsname` deve essere preallocato l'unico modo per farlo in maniera sicura è allora usare come dimensione il valore ottenuto con `sizeof(utsname)`.

Le informazioni vengono restituite in ciascuno dei singoli campi di `utsname` in forma di stringhe terminate dal carattere NUL. In particolare dette informazioni sono:

- il nome del sistema operativo;
- il nome della macchina (l'*hostname*);
- il nome della release del kernel;
- il nome della versione del kernel;
- il tipo di hardware della macchina;
- il nome del domino (il *domainname*);

ma l'ultima di queste informazioni è stata aggiunta di recente e non è prevista dallo standard POSIX, per questo essa è accessibile, come mostrato in fig. 6.1, solo se si è definita la macro `_GNU_SOURCE`.

Come accennato queste stesse informazioni, anche se a differenza di `sysctl` la funzione continua ad essere mantenuta, si possono ottenere direttamente tramite il filesystem `/proc`,

⁴con Linux ci sono in realtà 3 *system call* diverse per le dimensioni delle stringhe restituite, le prime due usano rispettivamente delle lunghezze di 9 e 65 byte, la terza usa anch'essa 65 byte, ma restituisce anche l'ultimo campo, `domainname`, con una lunghezza di 257 byte, la *glibc* provvede a mascherare questi dettagli usando la versione più recente disponibile.

esse infatti sono mantenute rispettivamente nei file `ostype`, `hostname`, `osrelease`, `version` e `domainname` che si trovano sotto la directory `/proc/sys/kernel/`.

6.2 La gestione del sistema

In questa sezione prenderemo in esame le interfacce di programmazione messe a disposizione per affrontare una serie di tematiche attinenti la gestione generale del sistema come quelle relative alla gestione di utenti e gruppi, al trattamento delle informazioni relative ai collegamenti al sistema, alle modalità per effettuare lo spegnimento o il riavvio di una macchina.

6.2.1 La gestione delle informazioni su utenti e gruppi

Tradizionalmente le informazioni utilizzate nella gestione di utenti e gruppi (password, corrispondenze fra nomi simbolici e *UID* numerici, home directory, ecc.) venivano registrate all'interno dei due file di testo `/etc/passwd` ed `/etc/group`, il cui formato è descritto dalle relative pagine del manuale⁵ e tutte le funzioni che richiedevano l'accesso a queste informazioni andavano a leggere direttamente il contenuto di questi file.

In realtà oltre a questi due file da molto tempo gran parte dei sistemi unix-like usano il cosiddetto sistema delle *shadow password* che prevede anche i due file `/etc/shadow` e `/etc/gshadow`, in cui sono state spostate le informazioni di autenticazione (ed inserite alcune estensioni di gestione avanzata) per toglierle dagli altri file che devono poter essere letti da qualunque processo per poter effettuare l'associazione fra username e *UID*.

Col tempo però questa impostazione ha incominciato a mostrare dei limiti. Da una parte il meccanismo classico di autenticazione è stato ampliato, ed oggi la maggior parte delle distribuzioni di GNU/Linux usa la libreria PAM (sigla che sta per *Pluggable Authentication Method*) che fornisce una interfaccia comune per i processi di autenticazione, svincolando completamente le singole applicazioni dai dettagli del come questa viene eseguita e di dove vengono mantenuti i dati relativi.

Si tratta di un sistema modulare, in cui è possibile utilizzare anche più meccanismi insieme, diventa così possibile avere vari sistemi di riconoscimento (biometria, chiavi hardware, ecc.), diversi formati per le password e diversi supporti per le informazioni. Il tutto avviene in maniera trasparente per le applicazioni purché per ciascun meccanismo si disponga della opportuna libreria che implementa l'interfaccia di PAM.

Dall'altra parte, il diffondersi delle reti e la necessità di centralizzare le informazioni degli utenti e dei gruppi per insiemi di macchine e servizi all'interno di una stessa organizzazione, in modo da mantenere coerenti i dati, ha portato anche alla necessità di poter recuperare e memorizzare dette informazioni su supporti diversi dai file citati, introducendo il sistema del *Name Service Switch*, che tratteremo brevemente in sez. 16.1.1 dato che la sua applicazione è cruciale nella procedura di risoluzione di nomi di rete.

In questo paragrafo ci limiteremo comunque a trattare le funzioni classiche per la lettura delle informazioni relative a utenti e gruppi tralasciando completamente quelle relative all'autenticazione.⁶ Le prime funzioni che vedremo sono quelle previste dallo standard POSIX.1; queste sono del tutto generiche e si appoggiano direttamente al *Name Service Switch*, per cui sono in grado di ricevere informazioni qualunque sia il supporto su cui esse vengono mantenu-

⁵nella quinta sezione, quella dei file di configurazione, dato che esistono comandi corrispondenti; per una trattazione sistematica dell'intero argomento coperto in questa sezione si consulti sez. 4.3 di [?].

⁶la cui programmazione ormai attiene all'uso dell'interfaccia di PAM, che va al di là dello scopo di questo testo.

te. Per leggere le informazioni relative ad un utente si possono usare due funzioni, `getpwuid` e `getpwnam`, i cui prototipi sono:

```
#include <pwd.h>
#include <sys/types.h>
struct passwd *getpwuid(uid_t uid)
struct passwd *getpwnam(const char *name)
```

Restituiscono le informazioni relative all'utente specificato.

Le funzioni ritornano il puntatore alla struttura contenente le informazioni in caso di successo e NULL nel caso non sia stato trovato nessun utente corrispondente a quanto specificato, nel qual caso `errno` assumerà il valore riportato dalle funzioni di sistema sottostanti.

Le due funzioni forniscono le informazioni memorizzate nel registro degli utenti (che nelle versioni più recenti per la parte di credenziali di autenticazione vengono ottenute attraverso PAM) relative all'utente specificato attraverso il suo *UID* o il nome di login. Entrambe le funzioni restituiscono un puntatore ad una struttura di tipo `passwd` la cui definizione (anch'essa eseguita in `pwd.h`) è riportata in fig. 6.2, dove è pure brevemente illustrato il significato dei vari campi.

```
struct passwd {
    char    *pw_name;          /* user name */
    char    *pw_passwd;        /* user password */
    uid_t   pw_uid;            /* user id */
    gid_t    pw_gid;           /* group id */
    char    *pw_gecos;         /* real name */
    char    *pw_dir;           /* home directory */
    char    *pw_shell;         /* shell program */
};
```

Figura 6.2: La struttura `passwd` contenente le informazioni relative ad un utente del sistema.

La struttura usata da entrambe le funzioni è allocata staticamente, per questo motivo viene sovrascritta ad ogni nuova invocazione, lo stesso dicasi per la memoria dove sono scritte le stringhe a cui i puntatori in essa contenuti fanno riferimento. Ovviamente questo implica che dette funzioni non possono essere rientranti; per questo motivo ne esistono anche due versioni alternative (denotate dalla solita estensione `_r`), i cui prototipi sono:

```
#include <pwd.h>
#include <sys/types.h>
struct passwd *getpwuid_r(uid_t uid, struct passwd *password, char *buffer,
                           size_t buflen, struct passwd **result)
struct passwd *getpwnam_r(const char *name, struct passwd *password, char *buffer,
                           size_t buflen, struct passwd **result)
```

Restituiscono le informazioni relative all'utente specificato.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà il valore riportato dalle di sistema funzioni sottostanti.

In questo caso l'uso è molto più complesso, in quanto bisogna prima allocare la memoria necessaria a contenere le informazioni. In particolare i valori della struttura `passwd` saranno restituiti all'indirizzo `password` mentre la memoria allocata all'indirizzo `buffer`, per un massimo di `buflen` byte, sarà utilizzata per contenere le stringhe puntate dai campi di `password`. Infine all'indirizzo puntato da `result` viene restituito il puntatore ai dati ottenuti, cioè `buffer` nel caso l'utente esista, o NULL altrimenti. Qualora i dati non possano essere contenuti nei

byte specificati da `buflen`, la funzione fallirà restituendo `ERANGE` (e `result` sarà comunque impostato a `NULL`).

Sia queste versioni rientranti che precedenti gli errori eventualmente riportati in `errno` in caso di fallimento dipendono dalla sottostanti funzioni di sistema usate per ricavare le informazioni (si veda quanto illustrato in sez. 6.5.1) per cui se lo si vuole utilizzare è opportuno inizializzarlo a zero prima di invocare le funzioni per essere sicuri di non avere un residuo di errore da una chiamata precedente. Il non aver trovato l'utente richiesto infatti può essere dovuto a diversi motivi (a partire dal fatto che non esista) per cui si possono ottenere i codici di errore più vari a seconda dei casi.

Del tutto analoghe alle precedenti sono le funzioni `getgrnam` e `getgrgid` che permettono di leggere le informazioni relative ai gruppi, i loro prototipi sono:

```
#include <grp.h>
#include <sys/types.h>
struct group *getgrgid(gid_t gid)
struct group *getgrnam(const char *name)
```

Restituiscono le informazioni relative al gruppo specificato.

Le funzioni ritornano il puntatore alla struttura contenente le informazioni in caso di successo e `NULL` nel caso non sia stato trovato nessun utente corrispondente a quanto specificato, nel qual caso `errno` assumerà il valore riportato dalle funzioni di sistema sottostanti.

Come per le precedenti per gli utenti esistono anche le analoghe versioni rientranti che di nuovo utilizzano la stessa estensione `_r`; i loro prototipi sono:

```
#include <grp.h>
#include <sys/types.h>
int getgrgid_r(gid_t gid, struct group *grp, char *buf, size_t buflen,
               struct group **result)
int getgrnam_r(const char *name, struct group *grp, char *buf, size_t buflen,
               struct group **result)
```

Restituiscono le informazioni relative al gruppo specificato.

Le funzioni ritornano 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà il valore riportato dalle funzioni di sistema sottostanti.

Il comportamento di tutte queste funzioni è assolutamente identico alle precedenti che leggono le informazioni sugli utenti, l'unica differenza è che in questo caso le informazioni vengono restituite in una struttura di tipo `group`, la cui definizione è riportata in fig. 6.3.

```
struct group {
    char    *gr_name;           /* group name */
    char    *gr_passwd;        /* group password */
    gid_t   gr_gid;            /* group id */
    char    **gr_mem;          /* group members */
};
```

Figura 6.3: La struttura `group` contenente le informazioni relative ad un gruppo del sistema.

Le funzioni viste finora sono in grado di leggere le informazioni sia direttamente dal file delle password in `/etc/passwd` che tramite il sistema del *Name Service Switch* e sono completamente generiche. Si noti però che non c'è una funzione che permetta di impostare direttamente una password.⁷ Dato che POSIX non prevede questa possibilità esiste un'altra interfaccia che lo fa,

⁷in realtà questo può essere fatto ricorrendo alle funzioni della libreria PAM, ma questo non è un argomento che tratteremo qui.

derivata da SVID le cui funzioni sono riportate in tab. 6.9. Questa interfaccia però funziona soltanto quando le informazioni sono mantenute su un apposito file di *registro* di utenti e gruppi, con il formato classico di `/etc/passwd` e `/etc/group`.

Funzione	Significato
<code>fgetpwent</code>	Legge una voce dal file di registro degli utenti specificato.
<code>fgetpwent_r</code>	Come la precedente, ma rientrante.
<code>putpwent</code>	Immette una voce in un file di registro degli utenti.
<code>getpwent</code>	Legge una voce da <code>/etc/passwd</code> .
<code>getpwent_r</code>	Come la precedente, ma rientrante.
<code>setpwent</code>	Ritorna all'inizio di <code>/etc/passwd</code> .
<code>endpwent</code>	Chiude <code>/etc/passwd</code> .
<code>fgetgrent</code>	Legge una voce dal file di registro dei gruppi specificato.
<code>fgetgrent_r</code>	Come la precedente, ma rientrante.
<code>putgrent</code>	Immette una voce in un file di registro dei gruppi.
<code>getgrent</code>	Legge una voce da <code>/etc/group</code> .
<code>getgrent_r</code>	Come la precedente, ma rientrante.
<code>setgrent</code>	Ritorna all'inizio di <code>/etc/group</code> .
<code>endgrent</code>	Chiude <code>/etc/group</code> .

Tabella 6.9: Funzioni per la manipolazione dei campi di un file usato come registro per utenti o gruppi nel formato di `/etc/passwd` e `/etc/group`.

Dato che oramai tutte le distribuzioni di GNU/Linux utilizzano le *shadow password* (quindi con delle modifiche rispetto al formato classico del file `/etc/passwd`), si tenga presente che le funzioni di questa interfaccia che permettono di scrivere delle voci in un *registro* degli utenti (cioè `putpwent` e `putgrent`) non hanno la capacità di farlo specificando tutti i contenuti necessari rispetto a questa estensione.

Per questo motivo l'uso di queste funzioni è deprecato, in quanto comunque non funzionale rispetto ad un sistema attuale, pertanto ci limiteremo a fornire soltanto l'elenco di tab. 6.9, senza nessuna spiegazione ulteriore. Chi volesse insistere ad usare questa interfaccia può fare riferimento alle pagine di manuale delle rispettive funzioni ed al manuale della *glibc* per i dettagli del funzionamento.

6.2.2 Il registro della *contabilità* degli utenti

Un altro insieme di funzioni relative alla gestione del sistema che esamineremo è quello che permette di accedere ai dati del registro della cosiddetta *contabilità* (o *accounting*) degli utenti. In esso vengono mantenute una serie di informazioni storiche relative sia agli utenti che si sono collegati al sistema, tanto per quelli correntemente collegati, che per la registrazione degli accessi precedenti, sia relative all'intero sistema, come il momento di lancio di processi da parte di `init`, il cambiamento dell'orologio di sistema, il cambiamento di `runlevel` o il riavvio della macchina.

I dati vengono usualmente memorizzati nei due file `/var/run/utmp` e `/var/log/wtmp`, che sono quelli previsti dal *Linux Filesystem Hierarchy Standard*, adottato dalla gran parte delle distribuzioni. Quando un utente si collega viene aggiunta una voce a `/var/run/utmp` in cui viene memorizzato il nome di login, il terminale da cui ci si collega, l'*UID* della shell di login, l'orario della connessione ed altre informazioni. La voce resta nel file fino al `logout`, quando viene cancellata e spostata in `/var/log/wtmp`.

In questo modo il primo file viene utilizzato per registrare chi sta utilizzando il sistema al momento corrente, mentre il secondo mantiene la registrazione delle attività degli utenti. A quest'ultimo vengono anche aggiunte delle voci speciali per tenere conto dei cambiamenti del

sistema, come la modifica del runlevel, il riavvio della macchina, ecc. Tutte queste informazioni sono descritte in dettaglio nel manuale della *glibc*.

Questi file non devono mai essere letti direttamente, ma le informazioni che contengono possono essere ricavate attraverso le opportune funzioni di libreria. Queste sono analoghe alle precedenti funzioni usate per accedere al registro degli utenti (vedi tab. 6.9), solo che in questo caso la struttura del registro della *contabilità* è molto più complessa, dato che contiene diversi tipi di informazione.

Le prime tre funzioni, `setutent`, `endutent` e `utmpname` servono rispettivamente a aprire e a chiudere il file che contiene il registro della *contabilità* degli, e a specificare su quale file esso viene mantenuto; i loro prototipi sono:

<code>#include <utmp.h></code>	
<code>void utmpname(const char *file)</code>	Specifica il file da usare come registro.
<code>void setutent(void)</code>	Apre il file del registro.
<code>void endutent(void)</code>	Chiude il file del registro.
Le funzioni non ritornano nulla.	

Si tenga presente che le funzioni non restituiscono nessun valore, pertanto non è possibile accorgersi di eventuali errori, ad esempio se si è impostato un nome di file sbagliato con `utmpname`.

Nel caso non si sia utilizzata `utmpname` per specificare un file di registro alternativo, sia `setutent` che `endutent` operano usando il default che è `/var/run/utmp` il cui nome, così come una serie di altri valori di default per i *pathname* di uso più comune, viene mantenuto nei valori di una serie di costanti definite includendo `paths.h`, in particolare quelle che ci interessano sono:

`_PATH_UTMP` specifica il file che contiene il registro per gli utenti correntemente collegati, questo è il valore che viene usato se non si è utilizzato `utmpname` per modificarlo;

`_PATH_WTMP` specifica il file che contiene il registro per l'archivio storico degli utenti collegati;

che nel caso di Linux hanno un valore corrispondente ai file `/var/run/utmp` e `/var/log/wtmp` citati in precedenza.

Una volta aperto il file del registro degli utenti si può eseguire una scansione leggendo o scrivendo una voce con le funzioni `getutent`, `getutid`, `getutline` e `pututline`, i cui prototipi sono:

<code>#include <utmp.h></code>	
<code>struct utmp *getutent(void)</code>	Legge una voce dalla posizione corrente nel registro.
<code>struct utmp *getutid(struct utmp *ut)</code>	Ricerca una voce sul registro.
<code>struct utmp *getutline(struct utmp *ut)</code>	Ricerca una voce sul registro attinente a un terminale.
<code>struct utmp *pututline(struct utmp *ut)</code>	Scrive una voce nel registro.
Le funzioni ritornano il puntatore ad una struttura <code>utmp</code> in caso di successo e <code>NULL</code> in caso di errore, nel qual caso <code>errno</code> assumerà il valore riportato dalle funzioni di sistema sottostanti.	

Tutte queste funzioni fanno riferimento ad una struttura di tipo `utmp`, la cui definizione in Linux è riportata in fig. 6.4. Le prime tre funzioni servono per leggere una voce dal registro: `getutent` legge semplicemente la prima voce disponibile, le altre due permettono di eseguire

una ricerca. Aprendo il registro con `setutent` ci si posiziona al suo inizio, ogni chiamata di queste funzioni eseguirà la lettura sulle voci seguenti, pertanto la posizione sulla voce appena letta, in modo da consentire una scansione del file. Questo vale anche per `getutid` e `getutline`, il che comporta che queste funzioni effettuano comunque una ricerca “*in avanti*”.

```

struct utmp
{
    short int ut_type;           /* Type of login. */
    pid_t ut_pid;               /* Process ID of login process. */
    char ut_line[UT_LINESIZE]; /* Devicename. */
    char ut_id[4];              /* Inittab ID. */
    char ut_user[UT_NAMESIZE];  /* Username. */
    char ut_host[UT_HOSTSIZE];  /* Hostname for remote login. */
    struct exit_status ut_exit; /* Exit status of a process marked
                                as DEAD_PROCESS. */

    long int ut_session;        /* Session ID, used for windowing. */
    struct timeval ut_tv;        /* Time entry was made. */
    int32_t ut_addr_v6[4];      /* Internet address of remote host. */
    char __unused[20];          /* Reserved for future use. */
};

```

Figura 6.4: La struttura `utmp` contenente le informazioni di una voce del registro di contabilità.

Con `getutid` si può cercare una voce specifica, a seconda del valore del campo `ut_type` dell'argomento `ut`. Questo può assumere i valori riportati in tab. 6.10, quando assume i valori `RUN_LVL`, `BOOT_TIME`, `OLD_TIME`, `NEW_TIME`, verrà restituito la prima voce che corrisponde al tipo determinato; quando invece assume i valori `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS` o `DEAD_PROCESS` verrà restituita la prima voce corrispondente al valore del campo `ut_id` specificato in `ut`.

Valore	Significato
EMPTY	Non contiene informazioni valide.
RUN_LVL	Identica il runlevel del sistema.
BOOT_TIME	Identifica il tempo di avvio del sistema.
OLD_TIME	Identifica quando è stato modificato l'orologio di sistema.
NEW_TIME	Identifica da quanto è stato modificato il sistema.
INIT_PROCESS	Identifica un processo lanciato da <code>init</code> .
LOGIN_PROCESS	Identifica un processo di login.
USER_PROCESS	Identifica un processo utente.
DEAD_PROCESS	Identifica un processo terminato.

Tabella 6.10: Classificazione delle voci del registro a seconda dei possibili valori del campo `ut_type`.

La funzione `getutline` esegue la ricerca sulle voci che hanno un `ut_type` con valore uguale a `LOGIN_PROCESS` o `USER_PROCESS`, restituendo la prima che corrisponde al valore di `ut_line`, che specifica il dispositivo di terminale che interessa, da indicare senza il `/dev/` iniziale. Lo stesso criterio di ricerca è usato da `pututline` per trovare uno spazio dove inserire la voce specificata; qualora questo spazio non venga trovato la voce viene aggiunta in coda al registro.

In generale occorre però tenere conto che queste funzioni non sono completamente standardizzate, e che in sistemi diversi possono esserci differenze; ad esempio `pututline` restituisce `void` in vari sistemi (compreso Linux, fino alle *libc5*). Qui seguiremo la sintassi fornita dalla *glibc*, ma gli standard POSIX 1003.1-2001 e XPG4.2 hanno introdotto delle nuove strutture

(e relativi file) di tipo `utmpx`, che sono un sovrainsieme della `utmp` usata tradizionalmente ed altrettante funzioni che le usano al posto di quelle citate.

La *glibc* utilizzava già una versione estesa di `utmp`, che rende inutili queste nuove strutture, per questo su Linux `utmpx` viene definita esattamente come `utmp`, con gli stessi campi di fig. 6.4. Altrettanto dicasi per le nuove funzioni di gestione previste dallo standard: `getutxent`, `getutxid`, `getutxline`, `pututxline`, `setutxent` e `endutxent`.

Tutte queste funzioni, definite con `utmpx` dal file di dichiarazione `utmpx.h`, su Linux sono ridefinite come sinonimi delle funzioni appena viste, con argomento di tipo `utmpx` anziché `utmp` ed hanno lo stesso identico comportamento. Per completezza viene definita anche `utmpxname` che non è prevista da POSIX.1-2001.

Come già visto in sez. 6.2.1, l'uso di strutture allocate staticamente rende le funzioni di lettura dei dati appena illustrate non rientranti. Per questo motivo la *glibc* fornisce anche delle versioni rientranti: `getutent_r`, `getutid_r`, `getutline_r`, che invece di restituire un puntatore restituiscono un intero e prendono due argomenti aggiuntivi, i rispettivi prototipi sono:

```
#include <utmp.h>
int *getutent_r(struct utmp *buffer, struct utmp **result)
                                     Legge una voce dalla posizione corrente nel registro.
int *getutid_r(struct utmp *buffer, struct utmp **result, struct utmp *ut)
                                     Ricerca una voce sul registro.
int *getutline_r(struct utmp *buffer, struct utmp **result, struct utmp *ut)
                                     Ricerca una voce sul registro attinente a un terminale.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà il valore riportato dalle funzioni di sistema sottostanti.

Le funzioni si comportano esattamente come le precedenti analoghe non rientranti, solo che restituiscono il risultato all'indirizzo specificato dal primo argomento aggiuntivo `buffer` mentre il secondo, `result`, viene usato per restituire il puntatore al buffer stesso.

Infine la *glibc* fornisce altre due funzioni, `updwtmp` e `logwtmp`, come estensione per scrivere direttamente delle voci nel file sul registro storico `/var/log/wtmp`; i rispettivi prototipi sono:

```
#include <utmp.h>
void updwtmp(const char *wtmp_file, const struct utmp *ut)
                                     Aggiunge una voce in coda al registro.
void logwtmp(const char *line, const char *name, const char *host)
                                     Aggiunge nel registro una voce con i valori specificati.
```

Le funzioni non restituiscono nulla.

La prima funzione permette l'aggiunta di una voce in coda al file del registro storico, indicato dal primo argomento, specificando direttamente una struttura `utmp`. La seconda invece utilizza gli argomenti `line`, `name` e `host` per costruire la voce che poi aggiunge chiamando `updwtmp`.

Queste funzioni non sono previste da POSIX.1-2001, anche se sono presenti in altri sistemi (ad esempio Solaris e NetBSD), per mantenere una coerenza con le altre funzioni definite nello standard che usano la struttura `utmpx` la *glibc* definisce anche una funzione `updwtmpx`, che come in precedenza è identica a `updwtmp` con la sola differenza di richiedere l'uso di `utmpx.h` e di una struttura `utmpx` come secondo argomento.

6.2.3 La gestione dello spegnimento e del riavvio

Una delle operazioni di gestione generale del sistema è quella che attiene alle modalità con cui se ne può gestire lo spegnimento ed il riavvio. Perché questo avvenga in maniera corretta, in particolare per le parti che comportano lo spegnimento effettivo della macchina, occorre che il

kernel effettui le opportune operazioni interagendo con il BIOS ed i dispositivi che controllano l'erogazione della potenza.

La funzione di sistema che controlla lo spegnimento ed il riavvio (ed altri aspetti della relativa procedura) è `reboot`,⁸ il cui prototipo è:

<pre>#include <unistd.h> #include <sys/reboot.h> int reboot(int cmd)</pre>	
Controlla il riavvio o l'arresto della macchina.	
La funzione non ritorna o ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
EFAULT	c'è un indirizzo non valido nel passaggio degli argomenti con il comando <code>LINUX_REBOOT_CMD_RESTART2</code> (obsoleto).
EINVAL	si sono specificati valori non validi per gli argomenti.
EPERM	il chiamante non ha i privilegi di amministratore (la <i>capability</i> <code>CAP_SYS_BOOT</code>).

La funzione, oltre al riavvio ed allo spegnimento, consente anche di controllare l'uso della combinazione di tasti tradizionalmente usata come scorciatoia da tastiera per richiedere il riavvio (`Ctrl-Alt-Del`, denominata in breve nella documentazione CAD) ed i suoi effetti specifici dipendono dalla architettura hardware. Se si è richiesto un riavvio o uno spegnimento in caso di successo la funzione, non esistendo più il programma, ovviamente non ritorna, pertanto bisogna avere cura di aver effettuato tutte le operazioni preliminari allo spegnimento prima di eseguirla.

Il comportamento della funzione viene controllato dall'argomento `cmd` e deve assumere indicato con una delle costanti seguente elenco, che illustra i comandi attualmente disponibili:

`LINUX_REBOOT_CMD_CAD_OFF`

Disabilita l'uso diretto della combinazione `Ctrl-Alt-Del`, la cui pressione si traduce nell'invio del segnale `SIGINT` a `init` (o più in generale al processo con `PID` 1) il cui effetto dipende dalla configurazione di quest'ultimo.

`LINUX_REBOOT_CMD_CAD_ON`

Attiva l'uso diretto della combinazione `Ctrl-Alt-Del`, la cui pressione si traduce nell'esecuzione dell'azione che si avrebbe avuto chiamando `reboot` con il comando `LINUX_REBOOT_CMD_RESTART`.

`LINUX_REBOOT_CMD_HALT`

Viene inviato sulla console il messaggio "*System halted.*" l'esecuzione viene bloccata immediatamente ed il controllo passato al monitor nella ROM (se esiste e l'architettura lo consente). Se non si è eseguita una sincronizzazione dei dati su disco con `sync` questi saranno perduti.

`LINUX_REBOOT_CMD_KEXEC`

viene eseguito direttamente il nuovo kernel che è stato opportunamente caricato in memoria da una `kexec_load` (che tratteremo a breve) eseguita in precedenza. La funzionalità è disponibile solo a partire dal kernel 2.6.13 e se il kernel corrente è stato compilato includendo il relativo supporto.⁹ Questo meccanismo consente di eseguire una sorta di riavvio rapido che evita di dover ripassare

⁸la funzione illustrata è quella fornita dalla *glibc* che maschera i dettagli di basso livello della *system call* la quale richiede attualmente tre argomenti; fino al kernel 2.1.30 la *system call* richiedeva un ulteriore quarto argomento, i primi due indicano dei *magic number* interi che possono assumere solo alcuni valori predefiniti, il terzo un comando, corrispondente all'unico argomento della funzione della *glibc* ed il quarto argomento aggiuntivo, ora ignorato, un puntatore generico ad ulteriori dati.

⁹deve essere stata abilitata l'opzione di compilazione `CONFIG_KEXEC`.

dalla inizializzazione da parte del BIOS ed il lancio del kernel attraverso un bootloader. Se non si è eseguita una sincronizzazione dei dati su disco con `sync` questi saranno perduti.

LINUX_REBOOT_CMD_POWER_OFF

Viene inviato sulla console il messaggio “*Power down.*” l’esecuzione viene bloccata immediatamente e la macchina, se possibile, viene spenta. Se non si è eseguita una sincronizzazione dei dati su disco con `sync` questi saranno perduti.

LINUX_REBOOT_CMD_RESTART

Viene inviato sulla console il messaggio “*Restarting system.*” ed avviata immediatamente la procedura di riavvio ordinaria. Se non si è eseguita una sincronizzazione dei dati su disco con `sync` questi saranno perduti.

LINUX_REBOOT_CMD_RESTART2

Viene inviato sulla console il messaggio “*Restarting system with command '%s'.*” ed avviata immediatamente la procedura di riavvio usando il comando fornito nell’argomento `arg` che viene stampato al posto di `%s` (veniva usato per lanciare un altro programma al posto di `init`). Nelle versioni recenti questo argomento viene ignorato ed il riavvio può essere controllato dall’argomento di avvio del kernel `reboot=...`. Se non si è eseguita una sincronizzazione dei dati su disco con `sync` questi saranno perduti.

Come appena illustrato usando il comando `LINUX_REBOOT_CMD_KEXEC` si può eseguire un riavvio immediato pre-caricando una immagine del kernel, che verrà eseguita direttamente. Questo meccanismo consente di evitare la reinizializzazione della macchina da parte del BIOS, ed oltre a velocizzare un eventuale riavvio, ha il vantaggio poter accedere allo stato corrente della macchina e della memoria, per cui viene usato spesso per installare un kernel di emergenza da eseguire in caso di crollo del sistema per recuperare il maggior numero di informazioni possibili.

La funzione di sistema che consente di caricare questa immagine del kernel è `kexec_load`, la funzione non viene definita nella `glibc` e deve pertanto essere invocata con `syscall`, il suo prototipo è:

```
#include <linux/kexec.h>
long kexec_load(unsigned long entry, unsigned long nr_segments, struct kexec_segment
                *segments, unsigned long flags)
```

Carica un kernel per un riavvio immediato.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBUSY</code>	c'è già un caricamento in corso, o un altro kernel è già in uso.
<code>EINVAL</code>	il valore di <code>flags</code> non è valido o si è indicato un valore eccessivo per <code>nr_segments</code> .
<code>EPERM</code>	il chiamante non ha i privilegi di amministratore (la <i>capability</i> <code>CAP_SYS_BOOT</code>).

Il primo argomento indica l’indirizzo fisico di esecuzione del nuovo kernel questo viene caricato usando un vettore di strutture `kexec_segment` (la cui definizione è riportata in fig. 6.5) che contengono i singoli segmenti dell’immagine. I primi due campi indicano indirizzo e dimensione del segmento di memoria in *user space*, i secondi indirizzo e dimensione in *kernel space*.

L’argomento `flags` è una maschera binaria contenente i flag che consentono di indicare le modalità con cui dovrà essere eseguito il nuovo kernel. La parte meno significativa viene usata per impostare l’architettura di esecuzione. Il valore `KEXEC_ARCH_DEFAULT` indica l’architettura

```
struct kexec_segment {
    void *buf;          /* Buffer in user space */
    size_t bufsz;        /* Buffer length in user space */
    void *mem;          /* Physical address of kernel */
    size_t memsz;        /* Physical address length */
};
```

Figura 6.5: La struttura kexec_segment per il caricamento di un segmento di immagine del kernel.

corrente, ma se ne può specificare anche una diversa, con i valori della seconda parte di tab. 6.11, e questa verrà usato posto che sia effettivamente eseguibile sul proprio processore.

Valore	Significato
KEXEC_ON_CRASH	Il kernel caricato sarà eseguito automaticamente in caso di crollo del sistema (dal kernel 2.6.13).
KEXEC_PRESERVE_CONTEXT	Viene preservato lo stato dei programmi e dei dispositivi prima dell'esecuzione del nuovo kernel. Viene usato principalmente per l'ibernazione del sistema ed ha senso solo se si è indicato un numero di segmento maggiore di zero (dal kernel 2.6.27).
KEXEC_ARCH_DEFAULT	Il kernel caricato verrà eseguito nella architettura corrente.
KEXEC_ARCH_XXX	Il kernel caricato verrà eseguito nella architettura indicata (con XXX che può essere: 386, X86_64, PPC, PPC64, IA_64, ARM, S390, SHMIPS e MIPS_LE).

Tabella 6.11: Valori per l'argomento flags di kexec_load.

I due valori più importanti sono però quelli della parte più significativa (riportati nella prima sezione di tab. 6.11). Il primo, KEXEC_ON_CRASH, consente di impostare l'esecuzione automatica del nuovo kernel caricato in caso di crollo del sistema, e viene usato quando si carica un kernel di emergenza da utilizzare per poter raccogliere informazioni diagnostiche che altrimenti verrebbero perdute non essendo il kernel ordinario più in grado di essere eseguito in maniera coerente. Il secondo valore, KEXEC_PRESERVE_CONTEXT, indica invece di preservare lo stato dei programmi e dei dispositivi, e viene in genere usato per realizzare la cosiddetta ibernazione in RAM.

6.3 Il controllo dell'uso delle risorse

Dopo aver esaminato in sez. 6.2 le funzioni che permettono di controllare le varie caratteristiche, capacità e limiti del sistema a livello globale, in questa sezione tratteremo le varie funzioni che vengono usate per quantificare le risorse (CPU, memoria, ecc.) utilizzate da ogni singolo processo e quelle che permettono di imporre a ciascuno di essi vincoli e limiti di utilizzo.

6.3.1 L'uso delle risorse

Come abbiamo accennato in sez. 3.1.5 le informazioni riguardo l'utilizzo delle risorse da parte di un processo è mantenuto in una struttura di tipo `rusage`, la cui definizione (che si trova in `sys/resource.h`) è riportata in fig. 6.6. Si ricordi che questa è una delle informazioni preservate attraverso una `exec`.

```

struct rusage {
    struct timeval ru_utime; /* user time used */
    struct timeval ru_stime; /* system time used */
    long ru_maxrss;          /* maximum resident set size */
    long ru_ixrss;           /* integral shared memory size */
    long ru_idrss;           /* integral unshared data size */
    long ru_isrss;           /* integral unshared stack size */
    long ru_minflt;          /* page reclaims */
    long ru_majflt;          /* page faults */
    long ru_nswap;           /* swaps */
    long ru_inblock;         /* block input operations */
    long ru_oublock;         /* block output operations */
    long ru_msgsnd;          /* messages sent */
    long ru_msgrcv;          /* messages received */
    long ru_nsignals;        /* signals received */
    long ru_nvcsw;           /* voluntary context switches */
    long ru_nivcsw;          /* involuntary context switches */
};

```

Figura 6.6: La struttura `rusage` per la lettura delle informazioni delle risorse usate da un processo.

La definizione della struttura in fig. 6.6 è ripresa da BSD 4.3,¹⁰ ma attualmente solo alcuni dei campi definiti sono effettivamente mantenuti. Con i kernel della serie 2.4 i soli campi che sono mantenuti sono: `ru_utime`, `ru_stime`, `ru_minflt` e `ru_majflt`. Con i kernel della serie 2.6 si aggiungono anche `ru_nvcsw` e `ru_nivcsw`, a partire dal 2.6.22 anche `ru_inblock` e `ru_oublock` e dal 2.6.32 anche `ru_maxrss`.

In genere includere esplicitamente `<sys/time.h>` non è più strettamente necessario, ma aumenta la portabilità, e serve comunque quando, come nella maggior parte dei casi, si debba accedere ai campi di `rusage` relativi ai tempi di utilizzo del processore, che sono definiti come strutture di tipo `timeval` (vedi fig. 4.15).

La struttura `rusage` è la struttura utilizzata da `wait4` (si ricordi quando visto in sez. 3.1.5) per ricavare la quantità di risorse impiegate dal processo di cui si è letto lo stato di terminazione, ma essa può anche essere letta direttamente utilizzando la funzione di sistema `getrusage`, il cui prototipo è:

<pre> #include <sys/time.h> #include <sys/resource.h> #include <unistd.h> int getrusage(int who, struct rusage *usage) </pre>	<p>Legge la quantità di risorse usate da un processo.</p>
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EINVAL l'argomento <code>who</code> non è valido</p> <p>ed inoltre EFAULT nel suo significato generico.</p>	

La funzione ritorna i valori per l'uso delle risorse nella struttura `rusage` puntata dall'argomento `usage`. L'argomento `who` permette di specificare il soggetto di cui si vuole leggere l'uso delle risorse; esso può assumere solo i valori illustrati in tab. 6.12, di questi `RUSAGE_THREAD` è specifico di Linux ed è disponibile solo a partire dal kernel 2.6.26. La funzione è stata recepi-

¹⁰questo non ha a nulla a che fare con il cosiddetto *BSD accounting* (vedi sez. 6.3.4) che si trova nelle opzioni di compilazione del kernel (e di norma è disabilitato) che serve per mantenere una contabilità delle risorse usate da ciascun processo in maniera molto più dettagliata.

ta nello standard POSIX.1-2001, che però indica come campi di `rusage` soltanto `ru_utime` e `ru_stime`.

Valore	Significato
RUSAGE_SELF	Ritorna l'uso delle risorse del processo corrente, che in caso di uso dei <i>thread</i> ammonta alla somma delle risorse utilizzate da tutti i <i>thread</i> del processo.
RUSAGE_CHILDREN	Ritorna l'uso delle risorse dell'insieme dei processi figli di cui è ricevuto lo stato di terminazione, che a loro volta comprendono quelle dei loro figli e così via.
RUSAGE_THREAD	Ritorna l'uso delle risorse del <i>thread</i> chiamante.

Tabella 6.12: Valori per l'argomento `who` di `getrusage`.

I campi più utilizzati sono comunque `ru_utime` e `ru_stime` che indicano rispettivamente il tempo impiegato dal processo nell'eseguire le istruzioni in *user space*, e quello impiegato dal kernel nelle *system call* eseguite per conto del processo (vedi sez. 6.4.1). I campi `ru_minflt` e `ru_majflt` servono a quantificare l'uso della memoria virtuale e corrispondono rispettivamente al numero di *page fault* (vedi sez. 2.2.1) avvenuti senza richiedere I/O su disco (i cosiddetti *minor page fault*), e a quelli che invece han richiesto I/O su disco (detti invece *major page fault*).

I campi `ru_nvcsw` e `runivcsw` indicano il numero di volte che un processo ha subito un *context switch* da parte dello *scheduler* rispettivamente nel caso un cui questo avviene prima dell'esaurimento della propria *time-slice* (in genere a causa di una *system call* bloccante), o per averla esaurita o essere stato interrotto da un processo a priorità maggiore. I campi `ru_inblock` e `ru_oublock` indicano invece il numero di volte che è stata eseguita una attività di I/O su un filesystem (rispettivamente in lettura e scrittura) ed infine `ru_maxrss` indica il valore più alto della *Resident Set Size* raggiunto dal processo stesso o, nel caso sia stato usato `RUSAGE_CHILDREN`, da uno dei suoi figli.

Si tenga conto che per un errore di implementazione nei i kernel precedenti il 2.6.9, nonostante questo fosse esplicitamente proibito dallo standard POSIX.1, l'uso di `RUSAGE_CHILDREN` comportava l'inserimento dell'ammontare delle risorse usate dai processi figli anche quando si era impostata una azione di `SIG_IGN` per il segnale `SIGCHLD` (per i segnali si veda cap. 7). Il comportamento è stato corretto per aderire allo standard a partire dal kernel 2.6.9.

6.3.2 Limiti sulle risorse

Come accennato nell'introduzione il kernel mette a disposizione delle funzionalità che permettono non solo di mantenere dati statistici relativi all'uso delle risorse, ma anche di imporre dei limiti precisi sul loro utilizzo da parte sia dei singoli processi che degli utenti.

Per far questo sono definite una serie di risorse e ad ogni processo vengono associati due diversi limiti per ciascuna di esse; questi sono il *limite corrente* (o *current limit*) che esprime un valore massimo che il processo non può superare ad un certo momento, ed il *limite massimo* (o *maximum limit*) che invece esprime il valore massimo che può assumere il *limite corrente*. In generale il primo viene chiamato anche *soft limit* dato che il suo valore può essere aumentato dal processo stesso durante l'esecuzione, ciò può però essere fatto solo fino al valore del secondo, che per questo viene detto *hard limit*.

In generale il superamento di un limite corrente comporta o l'emissione di uno specifico segnale o il fallimento della *system call* che lo ha provocato. A questo comportamento generico fanno eccezione `RLIMIT_CPU` in cui si ha in comportamento diverso per il superamento dei due

limiti e `RLIMIT_CORE` che influenza soltanto la dimensione o l'eventuale creazione dei file di *core dump* (vedi sez. 7.2.1).

Per permettere di leggere e di impostare i limiti di utilizzo delle risorse da parte di un processo sono previste due funzioni di sistema, `getrlimit` e `setrlimit`, i cui prototipi sono:

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrlimit(int resource, struct rlimit *rlim)
int setrlimit(int resource, const struct rlimit *rlim)
```

Legge i limiti di una risorsa.
Imposta i limiti di una risorsa.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL i valori per `resource` non sono validi o nell'impostazione si è specificato `rlim->rlim_cur` maggiore di `rlim->rlim_max`.

EPERM un processo senza i privilegi di amministratore ha cercato di innalzare i propri limiti.

ed inoltre **EFAULT** nel suo significato generico.

Entrambe le funzioni permettono di specificare attraverso l'argomento `resource` su quale risorsa si vuole operare. L'accesso (rispettivamente in lettura e scrittura) ai valori effettivi dei limiti viene poi effettuato attraverso la struttura `rlimit` puntata da `rlim`, la cui definizione è riportata in fig. 6.7, ed i cui campi corrispondono appunto a limite corrente e limite massimo.

```
struct rlimit {
    rlim_t    rlim_cur;    /* Soft limit */
    rlim_t    rlim_max;    /* Hard limit (ceiling for rlim_cur) */
};
```

Figura 6.7: La struttura `rlimit` per impostare i limiti di utilizzo delle risorse usate da un processo.

Come accennato un processo ordinario può alzare il proprio limite corrente fino al valore del limite massimo, e può anche ridurre, irreversibilmente, il valore di quest'ultimo. Nello specificare un limite, oltre a fornire dei valori specifici, si può anche usare la costante `RLIM_INFINITY` che permette di sbloccare completamente l'uso di una risorsa. Si ricordi però che solo un processo con i privilegi di amministratore¹¹ può innalzare un limite al di sopra del valore corrente del limite massimo ed usare un valore qualsiasi per entrambi i limiti.

Ciascuna risorsa su cui si possono applicare dei limiti è identificata da uno specifico valore dell'argomento `resource`, i valori possibili per questo argomento, ed il significato della risorsa corrispondente, dei rispettivi limiti e gli effetti causati dal superamento degli stessi sono riportati nel seguente elenco.

RLIMIT_AS Questa risorsa indica, in byte, la dimensione massima consentita per la memoria virtuale di un processo, il cosiddetto *Address Space*, (vedi sez. 2.2.1). Se il limite viene superato dall'uso di funzioni come `brk`, `mremap` o `mmap` esse falliranno con un errore di `ENOMEM`, mentre se il superamento viene causato dalla crescita dello *stack* il processo riceverà un segnale di `SIGSEGV`. Dato che il valore usato è un intero di tipo `long` nelle macchine a 32 bit questo può

¹¹per essere precisi in questo caso quello che serve è la *capability* `CAP_SYS_RESOURCE` (vedi sez. 9.1.1).

assumere un valore massimo di 2Gb (anche se la memoria disponibile può essere maggiore), in tal caso il limite massimo indicabile resta 2Gb, altrimenti la risorsa si dà per non limitata.

- RLIMIT_CORE Questa risorsa indica, in byte, la massima dimensione per un file di *core dump* (vedi sez. 7.2.1) creato nella terminazione di un processo. File di dimensioni maggiori verranno troncati a questo valore, mentre con un valore nullo si bloccherà la creazione dei *core dump*.
- RLIMIT_CPU Questa risorsa indica, in secondi, il massimo tempo di CPU (vedi sez. 6.4.2) che il processo può usare. Il superamento del limite corrente comporta l'emissione di un segnale di SIGXCPU, la cui azione predefinita (vedi sez. 7.2) è terminare il processo. Il segnale però può essere intercettato e ignorato, in tal caso esso verrà riemesso una volta al secondo fino al raggiungimento del limite massimo. Il superamento del limite massimo comporta comunque l'emissione di un segnale di SIGKILL. Si tenga presente che questo è il comportamento presente su Linux dai kernel della serie 2.2 ad oggi, altri kernel possono avere comportamenti diversi per quanto avviene quando viene superato il *soft limit*, pertanto per avere operazioni portabili è suggerito di intercettare sempre SIGXCPU e terminare in maniera ordinata il processo con la prima ricezione.
- RLIMIT_DATA Questa risorsa indica, in byte, la massima dimensione del segmento dati di un processo (vedi sez. 2.2.2). Il tentativo di allocare più memoria di quanto indicato dal limite corrente causa il fallimento della funzione di allocazione eseguita (*brk* o *sbrk* e dal kernel 4.7 anche *mmap*) con un errore di ENOMEM.
- RLIMIT_FSIZE Questa risorsa indica, in byte, la massima dimensione di un file che un processo può usare. Se il processo cerca di scrivere o di estendere il file oltre questa dimensione riceverà un segnale di SIGXFSZ, che di norma termina il processo. Se questo segnale viene intercettato la *system call* che ha causato l'errore fallirà con un errore di EFBIG.
- RLIMIT_LOCKS Questa risorsa indica il numero massimo di *file lock* (vedi sez. 10.1) e di *file lease* (vedi sez. 10.3.2) che un processo poteva effettuare. È un limite presente solo nelle prime versioni del kernel 2.4, pertanto non deve essere più utilizzato.
- RLIMIT_MEMLOCK Questa risorsa indica, in byte, l'ammontare massimo di memoria che può essere bloccata in RAM da un processo (vedi sez. 2.2.4). Dato che il *memory locking* viene effettuato sulle pagine di memoria, il valore indicato viene automaticamente arrotondato al primo multiplo successivo della dimensione di una pagina di memoria. Il limite comporta il fallimento delle *system call* che eseguono il *memory locking* (*mlock*, *mlockall* ed anche, vedi sez. 10.4.1, *mmap* con l'operazione MAP_LOCKED).
- Dal kernel 2.6.9 questo limite comprende anche la memoria che può essere bloccata da ciascun utente nell'uso della memoria condivisa (vedi sez. 11.2.6) con *shmctl*, che viene contabilizzata separatamente ma sulla quale viene applicato questo stesso limite. In precedenza invece questo limite veniva applicato sulla memoria condivisa per processi con privilegi amministrativi, il limite su questi è stato rimosso e la semantica della risorsa cambiata.
- RLIMIT_MSGQUEUE Questa risorsa indica il numero massimo di byte che possono essere utilizzati da un utente, identificato con l'*UID* reale del processo chiamante, per le code

di messaggi POSIX (vedi sez. 11.4.2). Per ciascuna coda che viene creata viene calcolata un'occupazione pari a:

```
bytes = attr.mq_maxmsg * sizeof(struct msg_msg *) +
        attr.mq_maxmsg * attr.mq_msgsize
```

dove `attr` è la struttura `mq_attr` (vedi fig. 11.30) usata nella creazione della coda. Il primo addendo consente di evitare la creazione di una coda con un numero illimitato di messaggi vuoti che comunque richiede delle risorse di gestione. Questa risorsa è stata introdotta con il kernel 2.6.8.

- RLIMIT_NICE** Questa risorsa indica il numero massimo a cui può essere portato il valore di *nice* (vedi sez. 3.3.2). Dato che non possono essere usati numeri negativi per specificare un limite, il valore di *nice* viene calcolato come `20-rlim_cur`. Questa risorsa è stata introdotta con il kernel 2.6.12.
- RLIMIT_NOFILE** Questa risorsa indica il numero massimo di file che un processo può aprire. Il tentativo di creazione di un ulteriore file descriptor farà fallire la funzione (`open`, `dup`, `pipe`, ecc.) con un errore `EMFILE`.
- RLIMIT_NPROC** Questa risorsa indica il numero massimo di processi che possono essere creati dallo stesso utente, che viene identificato con l'*UID* reale (vedi sez. 3.2.1) del processo chiamante. Se il limite viene raggiunto `fork` fallirà con un `EAGAIN`.
- RLIMIT_RSS** Questa risorsa indica, in pagine di memoria, la dimensione massima della memoria residente (il cosiddetto *RSS Resident Set Size*) cioè l'ammontare della memoria associata al processo che risiede effettivamente in RAM e non a quella eventualmente portata sulla *swap* o non ancora caricata dal filesystem per il segmento testo del programma. Ha effetto solo sulle chiamate a `madvise` con `MADV_WILLNEED` (vedi sez. 10.4.1). Presente solo sui i kernel precedenti il 2.4.30.
- RLIMIT_RTPRIO** Questa risorsa indica il valore massimo della priorità statica che un processo può assegnarsi o assegnare con `sched_setscheduler` e `sched_setparam` (vedi sez. 3.3.3). Il limite è stato introdotto a partire dal kernel 2.6.12 (ma per un bug è effettivo solo a partire dal 2.6.13). In precedenza solo i processi con privilegi amministrativi potevano avere una priorità statica ed utilizzare una politica di *scheduling* di tipo *real-time*.
- RLIMIT_RTTIME** Questa risorsa indica, in microsecondi, il tempo massimo di CPU che un processo eseguito con una priorità statica può consumare. Il superamento del limite corrente comporta l'emissione di un segnale di `SIGXCPU`, e quello del limite massimo di `SIGKILL` con le stesse regole viste `RLIMIT_CPU`: se `SIGXCPU` viene intercettato ed ignorato il segnale verrà rimesso ogni secondo fino al superamento del limite massimo. Questo limite è stato introdotto con il kernel 2.6.25 per impedire che un processo *real-time* possa bloccare il sistema.
- RLIMIT_SIGPENDING** Questa risorsa indica il numero massimo di segnali che possono essere mantenuti in coda per ciascun utente, identificato per *UID* reale. Il limite comprende sia i segnali normali che quelli *real-time* (vedi sez. 7.5.1) ed è attivo solo per `sigqueue`, con `kill` si potrà sempre inviare un segnale che non sia già presente su una coda. Questo limite è stato introdotto con il kernel 2.6.8.

RLIMIT_STACK Questa risorsa indica, in byte, la massima dimensione dello *stack* del processo. Se il processo esegue operazioni che estendano lo *stack* oltre questa dimensione riceverà un segnale di SIGSEGV.

A partire dal kernel 2.6.23 questo stesso limite viene applicato per la gran parte delle architetture anche ai dati che possono essere passati come argomenti e variabili di ambiente ad un programma posto in esecuzione con `execve`, nella misura di un quarto del valore indicato per lo *stack*. Questo valore in precedenza era fisso e pari a 32 pagine di memoria, corrispondenti per la gran parte delle architetture a 128kb di dati, dal 2.6.25, per evitare problemi di compatibilità quando **RLIMIT_STACK** è molto basso, viene comunque garantito uno spazio base di 32 pagine qualunque sia l'architettura.

Si tenga conto infine che tutti i limiti eventualmente presenti su un processo vengono ereditati dai figli da esso creati attraverso una `fork` (vedi sez. 3.1.3) e mantenuti invariati per i programmi messi in esecuzione attraverso una `exec` (vedi sez. 3.1.6).

Si noti come le due funzioni `getrlimit` e `setrlimit` consentano di operare solo sul processo corrente. Per questo motivo a partire dal kernel 2.6.36 (e dalla *glibc* 2.13) è stata introdotta un'altra funzione di sistema `prlimit` il cui scopo è quello di estendere e sostituire le precedenti; il suo prototipo è:

<pre>#include <sys/resource.h> int prlimit(pid_t pid, int resource, const struct rlimit *new_limit, struct rlimit *old_limit)</pre>	
	Legge e imposta i limiti di una risorsa.
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
EINVAL	i valori per <code>resource</code> non sono validi o nell'impostazione si è specificato <code>rlim->rlim_cur</code> maggiore di <code>rlim->rlim_max</code> .
EPERM	un processo senza i privilegi di amministratore ha cercato di innalzare i propri limiti o si è cercato di modificare i limiti di un processo di un altro utente.
ESRCH	il process <code>pid</code> non esiste.
ed inoltre EFAULT nel suo significato generico.	

La funzione è specifica di Linux e non portabile, per essere usata richiede che sia stata definita la macro `_GNU_SOURCE`. Il primo argomento indica il *PID* del processo di cui si vogliono cambiare i limiti e si può usare un valore nullo per indicare il processo chiamante. Per modificare i limiti di un altro processo, a meno di non avere privilegi amministrativi,¹² l'*UID* ed il *GID* reale del chiamante devono coincidere con *UID* e *GID* del processo indicato per i tre gruppi reale, effettivo e salvato.

Se `new_limit` non è `NULL` verrà usato come puntatore alla struttura `rlimit` contenente i valori dei nuovi limiti da impostare, mentre se `old_limit` non è `NULL` verranno letti i valori correnti dei limiti nella struttura `rlimit` da esso puntata. In questo modo è possibile sia leggere che scrivere, anche in contemporanea, i valori dei limiti. Il significato dell'argomento `resource` resta identico rispetto a `getrlimit` e `setrlimit`, così come i restanti requisiti.

6.3.3 Le informazioni sulle risorse di memoria e processore

La gestione della memoria è già stata affrontata in dettaglio in sez. 2.2; abbiamo visto allora che il kernel provvede il meccanismo della memoria virtuale attraverso la divisione della memoria fisica in pagine. In genere tutto ciò è del tutto trasparente al singolo processo, ma in certi

¹²anche in questo caso la *capability* necessaria è `CAP_SYS_RESOURCE` (vedi sez. 9.1.1).

casi, come per l'I/O mappato in memoria (vedi sez. 10.4.1) che usa lo stesso meccanismo per accedere ai file, è necessario conoscere le dimensioni delle pagine usate dal kernel. Lo stesso vale quando si vuole gestire in maniera ottimale l'interazione della memoria che si sta allocando con il meccanismo della paginazione.

Un tempo la dimensione delle pagine di memoria era fissata una volta per tutte dall'architettura hardware, per cui il relativo valore veniva mantenuto in una costante che bastava utilizzare in fase di compilazione. Oggi invece molte architetture permettono di variare questa dimensione (ad esempio sui PC recenti si possono usare pagine di 4kb e di 4 Mb) per cui per non dover ricompilare i programmi per ogni possibile caso e relativa scelta di dimensioni, è necessario poter utilizzare una funzione che restituisca questi valori quando il programma viene eseguito.

Dato che si tratta di una caratteristica generale del sistema come abbiamo visto in sez. 6.1 questa dimensione può essere ottenuta come tutte le altre attraverso una chiamata a `sysconf`, nel caso specifico si dovrebbe utilizzare il parametro `_SC_PAGESIZE`. Ma in BSD 4.2 è stata introdotta una apposita funzione di sistema `getpagesize` che restituisce la dimensione delle pagine di memoria. La funzione è disponibile anche su Linux (ma richiede che sia definita la macro `_BSD_SOURCE`) ed il suo prototipo è:

```
#include <unistd.h>
int getpagesize(void)
```

Legge la dimensione delle pagine di memoria.

La funzione ritorna la dimensione di una pagina in byte, e non sono previsti errori.

La funzione è prevista in SVr4, BSD 4.4 e SUSv2, anche se questo ultimo standard la etichetta come obsoleta, mentre lo standard POSIX 1003.1-2001 la ha eliminata, ed i programmi che intendono essere portabili devono ricorrere alla chiamata a `sysconf`.

In Linux è implementata come una *system call* nelle architetture in cui essa è necessaria, ed in genere restituisce il valore del simbolo `PAGE_SIZE` del kernel, che dipende dalla architettura hardware, anche se le versioni delle librerie del C precedenti la *glibc* 2.1 implementavano questa funzione restituendo sempre un valore statico.

La *glibc* fornisce, come specifica estensione GNU, altre due funzioni, `get_phys_pages` e `get_avphys_pages` che permettono di ottenere informazioni riguardo le pagine di memoria; i loro prototipi sono:

```
#include <sys/sysinfo.h>
long int get_phys_pages(void)
```

Legge il numero totale di pagine di memoria.

```
long int get_avphys_pages(void)
```

Legge il numero di pagine di memoria disponibili nel sistema.

Le funzioni ritornano il numero di pagine, e non sono previsti errori.

Queste funzioni sono equivalenti all'uso della funzione `sysconf` rispettivamente con i parametri `_SC_PHYS_PAGES` e `_SC_AVPHYS_PAGES`. La prima restituisce il numero totale di pagine corrispondenti alla RAM della macchina; la seconda invece la memoria effettivamente disponibile per i processi.

La *glibc* supporta inoltre, come estensioni GNU, due funzioni che restituiscono il numero di processori della macchina (e quello dei processori attivi); anche queste sono informazioni comunque ottenibili attraverso `sysconf` utilizzando rispettivamente i parametri `_SC_NPROCESSORS_CONF` e `_SC_NPROCESSORS_ONLN`.

Infine la *glibc* riprende da BSD la funzione `getloadavg` che permette di ottenere il carico di processore della macchina, in questo modo è possibile prendere decisioni su quando far partire eventuali nuovi processi, il suo prototipo è:

<pre>#include <stdlib.h> int getloadavg(double loadavg[], int nelem)</pre>	Legge il carico medio della macchina.
La funzione ritorna il numero di campionamenti restituiti e <code>-1</code> se non riesce ad ottenere il carico medio, <code>errno</code> non viene modificata.	

La funzione restituisce in ciascun elemento di `loadavg` il numero medio di processi attivi sulla coda dello *scheduler*, calcolato su diversi intervalli di tempo. Il numero di intervalli che si vogliono leggere è specificato da `nelem`, dato che nel caso di Linux il carico viene valutato solo su tre intervalli (corrispondenti a 1, 5 e 15 minuti), questo è anche il massimo valore che può essere assegnato a questo argomento.

6.3.4 La contabilità in stile BSD

Una ultima modalità per monitorare l'uso delle risorse è, se si è compilato il kernel con il relativo supporto,¹³ quella di attivare il cosiddetto *BSD accounting*, che consente di registrare su file una serie di informazioni¹⁴ riguardo alla contabilità delle risorse utilizzate da ogni processo che viene terminato.

Linux consente di salvare la contabilità delle informazioni relative alle risorse utilizzate dai processi grazie alla funzione `acct`, il cui prototipo è:

<pre>#include <unistd.h> int acct(const char *filename)</pre>	Abilita il <i>BSD accounting</i> .
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
EACCES non si hanno i permessi per accedere a <code>pathname</code> .	
ENOSYS il kernel non supporta il <i>BSD accounting</i> .	
EPERM il processo non ha privilegi sufficienti ad abilitare il <i>BSD accounting</i> .	
EUSERS non sono disponibili nel kernel strutture per il file o si è finita la memoria.	
ed inoltre EFAULT, EIO, ELOOP, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOTDIR, EROFS nel loro significato generico.	

La funzione attiva il salvataggio dei dati sul file indicato dal *pathname* contenuto nella stringa puntata da `filename`; la funzione richiede che il processo abbia i privilegi di amministratore (è necessaria la *capability* `CAP_SYS_PACCT`, vedi sez. 9.1.1). Se si specifica il valore `NULL` per `filename` il *BSD accounting* viene invece disabilitato. Un semplice esempio per l'uso di questa funzione è riportato nel programma `AcctCtrl.c` dei sorgenti allegati alla guida.

Quando si attiva la contabilità, il file che si indica deve esistere; esso verrà aperto in sola scrittura e le informazioni verranno registrate in *append* in coda al file tutte le volte che un processo termina. Le informazioni vengono salvate in formato binario, e corrispondono al contenuto della apposita struttura dati definita all'interno del kernel.

Il funzionamento di `acct` viene inoltre modificato da uno specifico parametro di sistema, modificabile attraverso `/proc/sys/kernel/acct` (o tramite la corrispondente `sysctl`). Esso contiene tre valori interi, il primo indica la percentuale di spazio disco libero sopra il quale viene ripresa una registrazione che era stata sospesa per essere scesi sotto il minimo indicato

¹³se cioè si è abilitata l'opzione di compilazione `CONFIG_BSD_PROCESS_ACCT`.

¹⁴contenute nella struttura `acct` definita nel file `include/linux/acct.h` dei sorgenti del kernel.

dal secondo valore (sempre in percentuale di spazio disco libero). Infine l'ultimo valore indica la frequenza in secondi con cui deve essere controllata detta percentuale.

6.4 La gestione dei tempi del sistema

In questa sezione, una volta introdotti i concetti base della gestione dei tempi da parte del sistema, tratteremo le varie funzioni attinenti alla gestione del tempo in un sistema unix-like, a partire da quelle per misurare i veri tempi di sistema associati ai processi, a quelle per convertire i vari tempi nelle differenti rappresentazioni che vengono utilizzate, a quelle della gestione di data e ora.

6.4.1 La misura del tempo in Unix

Tradizionalmente nei sistemi unix-like sono sempre stati previsti due tipi distinti di tempi, caratterizzati da altrettante modalità di misura ed espressi con diversi tipi di dati, chiamati rispettivamente *calendar time* e *process time*, secondo le seguenti definizioni:

calendar time

detto anche *tempo di calendario*, *tempo d'orologio* o *tempo reale*. Si tratta di un tempo assoluto o di un intervallo di tempo come lo intende normalmente per le misure fatte con un orologio. Per esprimere questo tempo è stato riservato il tipo `time_t`, e viene tradizionalmente misurato nel cosiddetto *unix-time*, espresso in secondi a partire dalla mezzanotte del primo gennaio 1970, data che viene chiamata *the Epoch*.

process time

detto anche *tempo di processore* o *tempo di CPU*. Si tratta del tempo impiegato da un processore nell'esecuzione del codice di un programma all'interno di un processo. Per esprimere questo tempo è stato riservato il tipo `clock_t`, e viene misurato nei cosiddetti *clock tick*, tradizionalmente corrispondenti al numero di interruzioni del processore da parte del timer di sistema. A differenza del precedente indica soltanto un intervallo di durata.

Il *calendar time* viene sempre mantenuto facendo riferimento al cosiddetto *tempo universale coordinato* UTC, anche se talvolta viene usato il cosiddetto GMT (*Greenwich Mean Time*) dato che l'UTC corrisponde all'ora locale di Greenwich. Si tratta del tempo su cui viene mantenuto il cosiddetto *orologio di sistema*, e viene usato per indicare i tempi dei file (quelli di sez. 4.3.4) o le date di avvio dei processi, ed è il tempo che viene usato dai demoni che compiono lavori amministrativi ad orari definito, come `cron`.

Si tenga presente che questo tempo è mantenuto dal kernel e non è detto che corrisponda al tempo misurato dall'orologio hardware presente su praticamente tutte le piastre madri dei computer moderni (il cosiddetto *hardware clock*), il cui valore viene gestito direttamente dall'hardware in maniera indipendente e viene usato dal kernel soltanto all'avvio per impostare un valore iniziale dell'orologio di sistema. La risoluzione tradizionale data dal tipo di dato `time_t` è di un secondo, ma nei sistemi più recenti sono disponibili altri tipi di dati con precisioni maggiori.

Si tenga presente inoltre che a differenza di quanto avviene con altri sistemi operativi,¹⁵ l'orologio di sistema viene mantenuto sempre in UTC e che la conversione all'ora locale del

¹⁵è possibile, ancorché assolutamente sconsigliabile, forzare l'orologio di sistema all'ora locale per compatibilità con quei sistemi operativi che han fatto questa deprecabile scelta.

proprio fuso orario viene effettuata dalle funzioni di libreria utilizzando le opportune informazioni di localizzazione (specificate in `/etc/timezone`). In questo modo si ha l'assicurazione che l'orologio di sistema misuri sempre un tempo monotono crescente come nella realtà, anche in presenza di cambi di fusi orari.

Il *process time* invece indica sempre una misura di un lasso di tempo e viene usato per tenere conto dei tempi di esecuzione dei processi. Esso viene sempre diviso in *user time* e *system time*, per misurare la durata di ciascun processo il kernel infatti calcola tre tempi:

- clock time* il tempo *reale*, viene chiamato anche *wall clock time* o *elapsed time*, passato dall'avvio del processo. Questo tempo fa riferimento al *calendar time* e dice la durata effettiva dell'esecuzione del processo, ma chiaramente dipende dal carico del sistema e da quanti altri processi stanno girando nello stesso momento.
- user time* il tempo effettivo che il processore ha impiegato nell'esecuzione delle istruzioni del programma in *user space*. È anche quello riportato nella risorsa `ru_utime` di `rusage` vista in sez. 6.3.1.
- system time* il tempo effettivo che il processore ha impiegato per eseguire codice delle *system call* nel kernel per conto del processo. È anche quello riportato nella risorsa `ru_stime` di `rusage` vista in sez. 6.3.1.

La somma di *user time* e *system time* indica il *process time*, vale a dire il tempo di processore totale che il sistema ha effettivamente utilizzato per eseguire il programma di un certo processo. Si può ottenere un riassunto dei valori di questi tempi quando si esegue un qualsiasi programma lanciando quest'ultimo come argomento del comando `time`.

Come accennato il *process time* viene misurato nei cosiddetti *clock tick*. Un tempo questo corrispondeva al numero di interruzioni effettuate dal timer di sistema, oggi lo standard POSIX richiede che esso sia espresso come multiplo della costante `CLOCKS_PER_SEC` che deve essere definita come 1000000, qualunque sia la risoluzione reale dell'orologio di sistema e la frequenza delle interruzioni del timer che, come accennato in sez. 3.1.1 e come vedremo a breve, è invece data dalla costante `HZ`.

Il tipo di dato usato per questo tempo, `clock_t`, con questa convenzione ha una risoluzione del microsecondo. Ma non tutte le funzioni di sistema come vedremo seguono questa convenzione, in tal caso il numero di *clock tick* al secondo può essere ricavato anche attraverso `sysconf` richiedendo il valore della costante `_SC_CLK_TCK` (vedi sez. 6.1.1). Il vecchio simbolo `CLK_TCK` definito in `time.h` è ormai considerato obsoleto e non deve essere usato.

In realtà tutti calcoli dei tempi vengono effettuati dal kernel per il cosiddetto *software clock*, utilizzando il *timer di sistema* e facendo i conti in base al numero delle interruzioni generate dello stesso, i cosiddetti “*jiffies*”. La durata di un “*jiffy*” è determinata dalla frequenza di interruzione del timer, indicata in Hertz, come accennato in sez. 3.1.1, dal valore della costante `HZ` del kernel, definita in `asm/param.h`.

Fino al kernel 2.4 il valore di `HZ` era 100 su tutte le architetture tranne l'alpha, per cui era 1000. Con il 2.6.0 è stato portato a 1000 su tutte le architetture, ma dal 2.6.13 il valore è diventato una opzione di compilazione del kernel, con un default di 250 e valori possibili di 100, 250, 1000. Dal 2.6.20 è stato aggiunto anche il valore 300 che è divisibile per le frequenze di refresh della televisione (50 o 60 Hz). Si può pensare che questi valori determinino anche la corrispondente durata dei *clock tick*, ma in realtà questa granularità viene calcolata in maniera indipendente usando la costante del kernel `USER_HZ`.

Fino al kernel 2.6.21 la durata di un *jiffy* costituiva la risoluzione massima ottenibile nella misura dei tempi impiegabile in una *system call* (ad esempio per i timeout). Con il

2.6.21 e l'introduzione degli *high-resolution timers* (HRT) è divenuto possibile ottenere, per le funzioni di attesa ed i timer, la massima risoluzione possibile fornita dall'hardware. Torneremo su questo in sez. 7.5.2.

6.4.2 La gestione del *process time*

Di norma tutte le operazioni del sistema fanno sempre riferimento al *calendar time*, l'uso del *process time* è riservato a quei casi in cui serve conoscere i tempi di esecuzione di un processo (ad esempio per valutarne l'efficienza). In tal caso infatti fare ricorso al *calendar time* è inutile in quanto il tempo può essere trascorso mentre un altro processo era in esecuzione o in attesa del risultato di una operazione di I/O.

La funzione più semplice per leggere il *process time* di un processo è `clock`, che da una valutazione approssimativa del tempo di CPU utilizzato dallo stesso; il suo prototipo è:

```
#include <time.h>
clock_t clock(void)
```

Legge il valore corrente del tempo di CPU.

La funzione ritorna il tempo di CPU in caso di successo e `-1` se questo non è ottenibile o rappresentabile in un valore di tipo `clock_t`, `errno` non viene usata.

La funzione restituisce il tempo in *clock tick* ma la *glibc* segue lo standard POSIX e quindi se si vuole il tempo in secondi occorre dividere il risultato per la costante `CLOCKS_PER_SEC`. In genere `clock_t` viene rappresentato come intero a 32 bit, il che comporta un valore massimo corrispondente a circa 72 minuti, dopo i quali il contatore riprenderà lo stesso valore iniziale.

La funzione è presente anche nello standard ANSI C, ma in tal caso non è previsto che il valore ritornato indichi un intervallo di tempo ma solo un valore assoluto, per questo se si vuole la massima portabilità anche al di fuori di kernel unix-like, può essere opportuno chiamare la funzione all'inizio del programma ed ottenere il valore del tempo con una differenza.

Si tenga presente inoltre che con altri kernel unix-like il valore riportato dalla funzione può includere anche il tempo di processore usato dai processi figli di cui si è ricevuto lo stato di terminazione con `wait` e affini. Questo non vale per Linux, in cui questa informazione deve essere ottenuta separatamente.

Come accennato in sez. 6.4.1 il tempo di processore è la somma di altri due tempi, l'*user time* ed il *system time*, che sono quelli effettivamente mantenuti dal kernel per ciascun processo. Questi possono essere letti separatamente attraverso la funzione `times`, il cui prototipo è:

```
#include <sys/times.h>
clock_t times(struct tms *buf)
```

Legge il valore corrente dei tempi di processore.

La funzione ritorna un numero di *clock tick* in caso di successo e `-1` per un errore, nel qual caso `errno` potrà assumere solo il valore `EFAULT` nel suo significato generico.

La funzione restituisce i valori di *process time* del processo corrente in una struttura di tipo `tms`, la cui definizione è riportata in fig. 6.8. La struttura prevede quattro campi; i primi due, `tms_utime` e `tms_stime`, sono l'*user time* ed il *system time* del processo, così come definiti in sez. 6.4.1. Gli altri due campi, `tms_cutime` e `tms_cstime`, riportano la somma dell'*user time* e del *system time* di tutti processi figli di cui si è ricevuto lo stato di terminazione.

Si tenga presente che i tempi di processore dei processi figli di un processo vengono sempre sommati al valore corrente ogni volta che se ne riceve lo stato di terminazione, e detto valore è quello che viene a sua volta ottenuto dal processo padre. Pertanto nei campi `tms_cutime`

```

struct tms {
    clock_t tms_utime; /* user time */
    clock_t tms_stime; /* system time */
    clock_t tms_cutime; /* user time of children */
    clock_t tms_cstime; /* system time of children */
};

```

Figura 6.8: La struttura `tms` dei tempi di processore associati a un processo.

e `tms_cstime` si sommano anche i tempi di ulteriori discendenti di cui i rispettivi genitori abbiano ricevuto lo stato di terminazione.

Si tenga conto che l'aggiornamento di `tms_cutime` e `tms_cstime` viene eseguito solo quando una chiamata a `wait` o `waitpid` è ritornata. Per questo motivo se un processo figlio termina prima di ricevere lo stato di terminazione di tutti i suoi figli, questi processi “nipoti” non verranno considerati nel calcolo di questi tempi e così via per i relativi “discendenti”.

Come accennato in sez. 6.3.1 per i kernel precedenti la versione 2.6.9 il tempo di processore dei processi figli veniva sommato comunque chiedendo di ignorare `SIGCHLD` anche se lo standard POSIX richiede esplicitamente che questo avvenga solo quando si riceve lo stato di uscita con una funzione della famiglia delle `wait`, anche in questo caso il comportamento è stato adeguato allo standard a partire dalla versione 2.6.9.

A differenza di quanto avviene per `clock` i valori restituiti nei campi di una struttura `tms` sono misurati in numero di *clock tick* effettivi e non in multipli di `CLOCKS_PER_SEC`, pertanto per ottenere il valore effettivo del tempo in secondi occorrerà dividere per il risultato di `sysconf(_SC_CLK_TCK)`.

Lo stesso vale per il valore di ritorno della funzione, il cui significato fa riferimento ad un tempo relativo ad un certo punto nel passato la cui definizione dipende dalle diverse implementazioni, e varia anche fra diverse versioni del kernel. Fino al kernel 2.4 si faceva infatti riferimento al momento dell'avvio del kernel. Con il kernel 2.6 si fa riferimento a $2^{32}/\text{HZ} - 300$ secondi prima dell'avvio.

Considerato che il numero dei *clock tick* per un kernel che è attivo da molto tempo può eccedere le dimensioni per il tipo `clock_t` il comportamento più opportuno per i programmi è di ignorare comunque il valore di ritorno della funzione e ricorrere alle funzioni per il tempo di calendario del prossimo paragrafo qualora si voglia calcolare il tempo effettivamente trascorso dall'inizio del programma.

Infine si tenga presente che per dei limiti nelle convenzioni per il ritorno dei valori delle *system call* su alcune architetture hardware (ed in particolare la `i386` dei PC a 32 bit) nel kernel della serie 2.6 il valore di ritorno della funzione può risultare erroneamente uguale a `-1`, indicando un errore, nei primi secondi dopo il boot (per la precisione nei primi 41 secondi) e se il valore del contatore eccede le dimensioni del tipo `clock_t`.

6.4.3 Le funzioni per il *calendar time*

Come anticipato in sez. 6.4.1 il *calendar time* viene espresso normalmente con una variabile di tipo `time_t`, che usualmente corrisponde ad un tipo elementare; in Linux è definito come `long int`, che di norma corrisponde a 32 bit, cosa che pone un limite al valore massimo esprimibile al 19 gennaio 2038, per ovviare alla cosa nelle versioni più recenti viene usato un valore di dimensioni maggiori. Il valore corrente del *calendar time*, che indicheremo come *tempo di*

sistema, può essere ottenuto con la funzione `time` che lo restituisce nel suddetto formato, il suo prototipo è:

```
#include <time.h>
time_t time(time_t *t)
```

Legge il valore corrente del *calendar time*.

La funzione ritorna il valore del *calendar time* in caso di successo e `-1` per un errore, nel qual caso `errno` potrà assumere solo il valore `EFAULT` nel suo significato generico.

L'argomento `t`, se non nullo, deve essere l'indirizzo di una variabile su cui duplicare il valore di ritorno, ma il suo uso è considerato obsoleto e deve essere sempre specificato come `NULL`, nel qual caso la funzione non può fallire.

Analoga a `time` è la funzione `stime` che serve per effettuare l'operazione inversa, e cioè per impostare il tempo di sistema qualora questo sia necessario; il suo prototipo è:

```
#include <time.h>
int stime(time_t *t)
```

Imposta il valore corrente del *calendar time*.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EPERM` non si hanno i permessi di amministrazione.
ed inoltre `EFAULT` nel suo significato generico.

Dato che modificare l'ora ha un impatto su tutto il sistema il cambiamento dell'orologio è una operazione privilegiata e questa funzione può essere usata solo da un processo con i privilegi di amministratore (per la precisione la *capability* `CAP_SYS_TIME`, vedi sez. 9.1.1), altrimenti la chiamata fallirà con un errore di `EPERM`.

Data la scarsa precisione nell'uso di `time_t`, che ha una risoluzione di un secondo, quando si devono effettuare operazioni sui tempi l'uso delle due funzioni precedenti è sconsigliato, ed esse sono di solito sostituite da `gettimeofday` e `settimeofday`,¹⁶ i cui prototipi sono:

```
#include <sys/time.h>
#include <time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz)
```

Legge il tempo corrente del sistema.

```
int settimeofday(const struct timeval *tv, const struct timezone *tz)
```

Imposta il tempo di sistema.

Le funzioni ritornano 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà i valori `EINVAL`, `EFAULT` e per `settimeofday` anche `EPERM`, nel loro significato generico.

Si noti come queste funzioni utilizzino per indicare il tempo una struttura di tipo `timeval`, la cui definizione si è già vista in fig. 4.15, questa infatti permette una espressione alternativa dei valori del *calendar time*, con una precisione, rispetto a `time_t`, fino al microsecondo, ma la precisione è solo teorica, e la precisione reale della misura del tempo dell'orologio di sistema non dipende dall'uso di queste strutture.

Come nel caso di `stime` anche `settimeofday` può essere utilizzata solo da un processo coi privilegi di amministratore e più precisamente con la capacità `CAP_SYS_TIME`. Si tratta comunque di una condizione generale che continua a valere per qualunque funzione che vada a modificare l'orologio di sistema, comprese tutte quelle che tratteremo in seguito.

Il secondo argomento di entrambe le funzioni è una struttura `timezone`, che storicamente veniva utilizzata per specificare appunto la cosiddetta *timezone*, cioè l'insieme del fuso orario

¹⁶Le due funzioni `time` e `stime` sono più antiche e derivano da SVr4, `gettimeofday` e `settimeofday` sono state introdotte da BSD, ed in BSD4.3 sono indicate come sostitute delle precedenti, `gettimeofday` viene descritta anche in POSIX.1-2001.

e delle convenzioni per l'ora legale che permettevano il passaggio dal tempo universale all'ora locale. Questo argomento oggi è obsoleto ed in Linux non è mai stato utilizzato; esso non è supportato né dalla vecchia *libc5*, né dalla *glibc*: pertanto quando si chiama questa funzione deve essere sempre impostato a `NULL`.

Modificare l'orologio di sistema con queste funzioni è comunque problematico, in quanto esse effettuano un cambiamento immediato. Questo può creare dei buchi o delle ripetizioni nello scorrere dell'orologio di sistema, con conseguenze indesiderate. Ad esempio se si porta avanti l'orologio si possono perdere delle esecuzioni di `cron` programmate nell'intervallo che si è saltato. Oppure se si porta indietro l'orologio si possono eseguire due volte delle operazioni previste nell'intervallo di tempo che viene ripetuto.

Per questo motivo la modalità più corretta per impostare l'ora è quella di usare la funzione `adjtime`, il cui prototipo è:

<pre>#include <sys/time.h> int adjtime(const struct timeval *delta, struct timeval *olddelta)</pre> <p style="text-align: right;">Aggiusta l'orologio di sistema.</p>
<p>La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EINVAL</code> il valore di <code>delta</code> eccede il massimo consentito.</p> <p><code>EPERM</code> il processo non i privilegi di amministratore.</p>

Questa funzione permette di avere un aggiustamento graduale del tempo di sistema in modo che esso sia sempre crescente in maniera monotona. Il valore indicato nella struttura `timeval` puntata da `delta` esprime il valore di cui si vuole spostare l'orologio. Se è positivo l'orologio sarà accelerato per un certo tempo in modo da guadagnare il tempo richiesto, altrimenti sarà rallentato.

La funzione è intesa per piccoli spostamenti del tempo di sistema, ed esistono pertanto dei limiti massimi per i valori che si possono specificare per `delta`. La *glibc* impone un intervallo compreso fra `INT_MIN/1000000 + 2` e `INT_MAX/1000000 - 2`, corrispondente, su una architettura PC ordinaria a 32 bit, ad un valore compreso fra `-2145` e `2145` secondi.

Inoltre se si invoca la funzione prima che una precedente richiesta di aggiustamento sia stata completata, specificando un altro valore, il precedente aggiustamento viene interrotto, ma la parte dello stesso che è già stata completata non viene rimossa. Però è possibile in questo caso farsi restituire nella struttura puntata da `olddelta` il tempo restante della precedente richiesta. Fino al kernel 2.6.26 ed alla *glibc* 2.8 questo però era possibile soltanto specificando un diverso aggiustamento per `delta`, il bug è stato corretto a partire dalle versioni citate e si può ottenere l'informazione relativa alla frazione di aggiustamento mancante usando il valore `NULL` per `delta`.

Linux poi prevede una specifica funzione di sistema che consente un aggiustamento molto più dettagliato del tempo, permettendo ad esempio anche di regolare anche la velocità e le derivate dell'orologio di sistema. La funzione è `adjtimex` ed il suo prototipo è:

<pre>#include <sys/timex.h> int adjtimex(struct timex *buf)</pre> <p style="text-align: right;">Regola l'orologio di sistema.</p>
<p>La funzione ritorna lo stato dell'orologio (un valore ≥ 0) in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EINVAL</code> si sono indicati valori fuori dall'intervallo consentito per qualcuno dei campi di <code>buf</code>.</p> <p><code>EPERM</code> si è richiesta una modifica dei parametri ed il processo non ha i privilegi di amministratore.</p> <p>ed inoltre <code>EFAULT</code> nel suo significato generico.</p>

In caso di successo la funzione restituisce un valore numerico non negativo che indica lo stato dell'orologio, che può essere controllato con i valori delle costanti elencate in tab. 6.13.

Nome	Valore	Significato
TIME_OK	0	Orologio sincronizzato.
TIME_INS	1	Inserimento di un <i>leap second</i> .
TIME_DEL	2	Cancellazione di un <i>leap second</i> .
TIME_OOP	3	<i>leap second</i> in corso.
TIME_WAIT	4	<i>leap second</i> avvenuto.
TIME_BAD	5	Orologio non sincronizzato.

Tabella 6.13: Possibili valori ritornati da `adjtimex` in caso di successo.

La funzione richiede come argomento il puntatore ad una struttura di tipo `timex`, la cui definizione, effettuata in `sys/timex.h`, è riportata in fig. 6.9 per i campi che interessano la possibilità di essere modificati. In realtà la struttura è stata estesa con ulteriori campi, i cui valori sono utilizzabili solo in lettura, non riportati in fig. 6.9, i dettagli di questi campi si possono recuperare dalla pagina di manuale di `adjtimex`.

```

struct timex {
    int modes;           /* mode selector */
    long offset;         /* time offset (usec) */
    long freq;          /* frequency offset (scaled ppm) */
    long maxerror;       /* maximum error (usec) */
    long esterror;       /* estimated error (usec) */
    int status;          /* clock command/status */
    long constant;       /* pll time constant */
    long precision;      /* clock precision (usec) (read only) */
    long tolerance;      /* clock frequency tolerance (ppm) (read only) */
    struct timeval time; /* (read only) */
    long tick;           /* (modified) usecs between clock ticks */
    /* ulteriori campi accessibili in sola lettura... */
    ...
};

```

Figura 6.9: La struttura `timex` per il controllo dell'orologio di sistema.

L'azione della funzione dipende dal valore del campo `mode` di `buf`, che specifica quale parametro dell'orologio di sistema, specificato nel corrispondente campo di `timex`, deve essere impostato. Un valore nullo serve per leggere i parametri correnti, i valori diversi da zero devono essere specificati come OR binario delle costanti riportate in tab. 6.14.

La funzione utilizza il meccanismo di David L. Mills, descritto nell'RFC 1305, che è alla base del protocollo NTP. La funzione è specifica di Linux e non deve essere usata se la portabilità è un requisito, la `glibc` provvede anche un suo omonimo `ntp_adjtime`. La trattazione completa di questa funzione necessita di una lettura approfondita del meccanismo descritto nell'RFC 1305, ci limitiamo a descrivere in tab. 6.14 i principali valori utilizzabili per il campo `mode`, un elenco più dettagliato del significato dei vari campi della struttura `timex` può essere ritrovato in [?].

Il valore delle costanti per `mode` può essere anche espresso, secondo la sintassi specificata per la forma equivalente di questa funzione definita come `ntp_adjtime`, utilizzando il prefisso `MOD` al posto di `ADJ`.

Nome	Significato
ADJ_OFFSET	Imposta la differenza fra il tempo reale e l'orologio di sistema: deve essere indicata in microsecondi nel campo <code>offset</code> di <code>timex</code> .
ADJ_FREQUENCY	Imposta la differenza in frequenza fra il tempo reale e l'orologio di sistema: deve essere indicata in parti per milione nel campo <code>frequency</code> di <code>timex</code> .
ADJ_MAXERROR	Imposta il valore massimo dell'errore sul tempo, espresso in microsecondi nel campo <code>maxerror</code> di <code>timex</code> .
ADJ_ESTERROR	Imposta la stima dell'errore sul tempo, espresso in microsecondi nel campo <code>esterror</code> di <code>timex</code> .
ADJ_STATUS	Imposta alcuni valori di stato interni usati dal sistema nella gestione dell'orologio specificati nel campo <code>status</code> di <code>timex</code> .
ADJ_TIMECONST	Imposta la larghezza di banda del PLL implementato dal kernel, specificato nel campo <code>constant</code> di <code>timex</code> .
ADJ_TICK	Imposta il valore dei <i>tick</i> del timer in microsecondi, espresso nel campo <code>tick</code> di <code>timex</code> .
ADJ_OFFSET_SINGLESOT	Chiede uno spostamento <i>una tantum</i> dell'orologio secondo il valore del campo <code>offset</code> simulando il comportamento di <code>adjtime</code> .

Tabella 6.14: Costanti per l'assegnazione del valore del campo `mode` della struttura `timex`.

Si tenga presente infine che con l'introduzione a partire dal kernel 2.6.21 degli *high-resolution timer* ed il supporto per i cosiddetti POSIX *real-time clock*, si può ottenere il *calendar time* direttamente da questi, come vedremo in sez. 7.5.2, con la massima risoluzione possibile per l'hardware della macchina.

6.4.4 La gestione delle date.

Le funzioni viste al paragrafo precedente sono molto utili per trattare le operazioni elementari sui tempi, però le rappresentazioni del tempo ivi illustrate, se han senso per specificare un intervallo, non sono molto intuitive quando si deve esprimere un'ora o una data. Per questo motivo è stata introdotta una ulteriore rappresentazione, detta *broken-down time*, che permette appunto di *suddividere* il *calendar time* usuale in ore, minuti, secondi, ecc. e viene usata tenendo conto anche dell'eventuale utilizzo di un fuso orario.

```

struct tm {
    int      tm_sec;          /* seconds */
    int      tm_min;          /* minutes */
    int      tm_hour;         /* hours */
    int      tm_mday;         /* day of the month */
    int      tm_mon;          /* month */
    int      tm_year;         /* year */
    int      tm_wday;         /* day of the week */
    int      tm_yday;         /* day in the year */
    int      tm_isdst;        /* daylight saving time */
    long int tm_gmtoff;       /* Seconds east of UTC. */
    const char *tm_zone;     /* Timezone abbreviation. */
};

```

Figura 6.10: La struttura `tm` per una rappresentazione del tempo in termini di ora, minuti, secondi, ecc.

Questo viene effettuato attraverso una opportuna struttura `tm`, la cui definizione è riportata in fig. 6.10, ed è in genere questa struttura che si utilizza quando si deve specificare un tempo a partire dai dati naturali (ora e data), dato che essa consente anche di tenere conto della gestione del fuso orario e dell'ora legale. In particolare gli ultimi due campi, `tm_gmtoff` e `tm_zone`, sono estensioni previste da BSD e supportate dalla *glibc* quando è definita la macro `_BSD_SOURCE`.

Ciascuno dei campi di `tm` ha dei precisi intervalli di valori possibili, con convenzioni purtroppo non troppo coerenti. Ad esempio `tm_sec` che indica i secondi deve essere nell'intervallo da 0 a 59, ma è possibile avere anche il valore 60 per un cosiddetto *leap second* (o *secondo intercalare*), cioè uno di quei secondi aggiunti al calcolo dell'orologio per effettuare gli aggiustamenti del calendario per tenere conto del disallineamento con il tempo solare.¹⁷

I campi `tm_min` e `tm_hour` che indicano rispettivamente minuti ed ore hanno valori compresi rispettivamente fra 0 e 59 e fra 0 e 23. Il campo `tm_mday` che indica il giorno del mese prevede invece un valore compreso fra 1 e 31, ma la *glibc* supporta pure il valore 0 come indicazione dell'ultimo giorno del mese precedente. Il campo `tm_mon` indica il mese dell'anno a partire da gennaio con valori compresi fra 0 e 11.

I campi `tm_wday` e `tm_yday` indicano invece rispettivamente il giorno della settimana, a partire dalla Domenica, ed il giorno dell'anno, a partire del primo gennaio, ed hanno rispettivamente valori compresi fra 0 e 6 e fra 0 e 365. L'anno espresso da `tm_year` viene contato come numero di anni a partire dal 1900. Infine `tm_isdst` è un valore che indica se per gli altri campi si intende come attiva l'ora legale ed influenza il comportamento di `mktime`.

Le funzioni per la gestione del *broken-down time* sono varie e vanno da quelle usate per convertire gli altri formati in questo, usando o meno l'ora locale o il tempo universale, a quelle per trasformare il valore di un tempo in una stringa contenente data ed ora. Le prime due funzioni, `asctime` e `ctime` servono per poter stampare in forma leggibile un tempo, i loro prototipi sono:

<pre>#include <time.h> char * asctime(const struct tm *tm) char * ctime(const time_t *timep)</pre>	<p>Convertite un <i>broken-down time</i> in una stringa.</p> <p>Convertite un <i>calendar time</i> in una stringa.</p>
<p>Le funzioni ritornano un puntatore al risultato in caso di successo e NULL per un errore, <code>errno</code> non viene modificata.</p>	

Le funzioni prendono rispettivamente come argomenti i puntatori ad una struttura `tm` contenente un *broken-down time* o ad una variabile di tipo `time_t` che esprime il *calendar time*, restituendo il puntatore ad una stringa che esprime la data, usando le abbreviazioni standard di giorni e mesi in inglese, nella forma:

Sun Apr 29 19:47:44 2012\n"

Nel caso di `ctime` la funzione tiene conto della eventuale impostazione di una *timezone* e effettua una chiamata preventiva a `tzset` (che vedremo a breve), in modo che la data espressa tenga conto del fuso orario. In realtà `ctime` è banalmente definita in termini di `asctime` come `asctime(localtime(t))`.

Dato che l'uso di una stringa statica rende le funzioni non rientranti POSIX.1c e SUSv2 prevedono due sostitute rientranti, il cui nome è al solito ottenuto aggiungendo un `_r`, che prendono un secondo argomento `char *buf`, in cui l'utente deve specificare il buffer su cui la stringa deve essere copiata (deve essere di almeno 26 caratteri).

¹⁷per dettagli si consulti http://it.wikipedia.org/wiki/Leap_second.

Per la conversione fra *broken-down time* e *calendar time* sono invece disponibili altre tre funzioni, `gmtime`, `localtime` e `mktime` i cui prototipi sono:

```
struct tm * gmtime(const time_t *timep)
    Converta un calendar time in un broken-down time in UTC.
struct tm * localtime(const time_t *timep)
    Converta un calendar time in un broken-down time nell'ora locale.
time_t mktime(struct tm *tm)
    Converta un broken-down time in un calendar time.
```

Le funzioni ritornano un puntatore al risultato in caso di successo e NULL per un errore, tranne che `mktime` che restituisce direttamente il valore o `-1` in caso di errore, `errno` non viene modificata.

Le prime funzioni, `gmtime`, `localtime` servono per convertire il tempo in *calendar time* specificato da un argomento di tipo `time_t` restituendo un *broken-down time* con il puntatore ad una struttura `tm`. La prima effettua la conversione senza tenere conto del fuso orario, esprimendo la data in tempo coordinato universale (UTC), cioè l'ora di Greenwich, mentre `localtime` usa l'ora locale e per questo effettua una chiamata preventiva a `tzset`.

Anche in questo caso le due funzioni restituiscono l'indirizzo di una struttura allocata staticamente, per questo sono state definite anche altre due versioni rientranti (con la solita estensione `_r`), che prevedono un secondo argomento `struct tm *result`, fornito dal chiamante, che deve preallocare la struttura su cui sarà restituita la conversione. La versione rientrante di `localtime` però non effettua la chiamata preventiva a `tzset` che deve essere eseguita a cura dell'utente.

La funzione `mktime` esegue invece la conversione di un *broken-down time* a partire da una struttura `tm` restituendo direttamente un valore di tipo `time_t` con il *calendar time*. La funzione ignora i campi `tm_wday` e `tm_yday` e per gli altri campi normalizza eventuali valori fuori degli intervalli specificati in precedenza: se cioè si indica un 12 per `tm_mon` si prenderà il gennaio dell'anno successivo.

Inoltre la funzione tiene conto del valore di `tm_isdst` per effettuare le correzioni relative al fuso orario: un valore positivo indica che deve essere tenuta in conto l'ora legale, un valore nullo che non deve essere applicata nessuna correzione, un valore negativo che si deve far ricorso alle informazioni relative al proprio fuso orario per determinare lo stato dell'ora legale.

La funzione infine modifica i valori della struttura `tm` in forma di *value result argument*, normalizzando i valori dei vari campi, impostando i valori risultanti per `tm_wday` e `tm_yday` e assegnando a `tm_isdst` il valore (positivo o nullo) corrispondente allo stato dell'ora legale. La funzione provvede anche ad impostare il valore della variabile globale `tzname`.

Come accennato l'uso del *broken-down time* permette di tenere conto anche della differenza fra tempo universale e ora locale, compresa l'eventuale ora legale. Questo viene fatto dalle funzioni di conversione grazie alle informazioni riguardo la propria *timezone* mantenute nelle tre variabili globali mostrate in fig. 6.11, cui si può accedere direttamente includendo `time.h`. Come illustrato queste variabili vengono impostate internamente da alcune delle precedenti funzioni di conversione, ma lo si può fare esplicitamente chiamando direttamente la funzione `tzset`, il cui prototipo è:

```
#include <time.h>
void tzset(void)
    Imposta le variabili globali della timezone.
```

La funzione non ritorna niente e non dà errori.

La funzione inizializza le variabili di fig. 6.11 a partire dal valore della variabile di ambiente `TZ`, se quest'ultima non è definita verrà usato il file `/etc/localtime`. La variabile `tzname`

contiene due stringhe, che indicano i due nomi standard della *timezone* corrente. La prima è il nome per l'ora solare, la seconda per l'ora legale. Anche se in fig. 6.11 sono indicate come `char *` non è il caso di modificare queste stringhe. La variabile `timezone` indica la differenza di fuso orario in secondi, mentre `daylight` indica se è attiva o meno l'ora legale.

```
extern char *tzname[2];
extern long timezone;
extern int daylight;
```

Figura 6.11: Le variabili globali usate per la gestione delle *timezone*.

Benché la funzione `asctime` fornisca la modalità più immediata per stampare un tempo o una data, la flessibilità non fa parte delle sue caratteristiche; quando si vuole poter stampare solo una parte (l'ora, o il giorno) di un tempo si può ricorrere alla più sofisticata `strftime`, il cui prototipo è:

<pre>#include <time.h> size_t strftime(char *s, size_t max, const char *format, const struct tm *tm)</pre> <p>Crea una stringa con una data secondo il formato indicato.</p>
<p>La funzione ritorna il numero di caratteri inseriti nella stringa <code>s</code> oppure 0, <code>errno</code> non viene modificata.</p>

La funzione converte il *broken-down time* indicato nella struttura puntata dall'argomento `tm` in una stringa di testo da salvare all'indirizzo puntato dall'argomento `s`, purché essa sia di dimensione inferiore al massimo indicato dall'argomento `max`. Il numero di caratteri generati dalla funzione viene restituito come valore di ritorno, senza tener però conto del terminatore finale, che invece viene considerato nel computo della dimensione. Se quest'ultima è eccessiva viene restituito 0 e lo stato di `s` è indefinito.

Il risultato della funzione è controllato dalla stringa di formato `format`, tutti i caratteri restano invariati eccetto % che viene utilizzato come modificatore. Alcuni dei possibili valori che esso può assumere sono riportati in tab. 6.15.¹⁸ La funzione tiene conto anche delle eventuali impostazioni di localizzazione per stampare i vari nomi in maniera adeguata alla lingua scelta, e con le convenzioni nazionali per i formati di data ed ora.

Infine per effettuare l'operazione di conversione inversa, da una stringa ad un *broken-down time*, si può utilizzare la funzione `strptime`, il cui prototipo è:

<pre>#include <time.h> char *strptime(const char *s, const char *format, struct tm *tm)</pre> <p>Converte una stringa con in un <i>broken-down time</i> secondo un formato.</p>
<p>La funzione ritorna il puntatore al primo carattere non processato della stringa o al terminatore finale qualora questa sia processata interamente, <code>errno</code> non viene modificata.</p>

La funzione processa la stringa puntata dall'argomento `s` da sinistra a destra, utilizzando il formato contenuto nella stringa puntata dall'argomento `format`, avvalorando volta volta i corrispondenti campi della struttura puntata dall'argomento `tm`. La scansione si interrompe immediatamente in caso di mancata corrispondenza a quanto indicato nella stringa di formato, che usa una sintassi analoga a quella già vista per `strftime`. La funzione supporta i

¹⁸per la precisione si sono riportati quelli definiti dallo standard ANSI C che sono anche quelli ripresi in POSIX.1; la *glibc* fornisce anche le estensioni introdotte da POSIX.2 per il comando `date`, i valori introdotti da SVID3 e ulteriori estensioni GNU; l'elenco completo dei possibili valori è riportato nella pagina di manuale della funzione.

Modificatore	Esempio	Significato
%a	Wed	Nome del giorno, abbreviato.
%A	Wednesday	Nome del giorno, completo.
%b	Apr	Nome del mese, abbreviato.
%B	April	Nome del mese, completo.
%c	Wed Apr 24 18:40:50 2002	Data e ora.
%d	24	Giorno del mese.
%H	18	Ora del giorno, da 0 a 24.
%I	06	Ora del giorno, da 0 a 12.
%j	114	Giorno dell'anno.
%m	04	Mese dell'anno.
%M	40	Minuto.
%p	PM	AM/PM.
%S	50	Secondo.
%U	16	Settimana dell'anno (partendo dalla domenica).
%w	3	Giorno della settimana.
%W	16	Settimana dell'anno (partendo dal lunedì).
%x	04/24/02	La data.
%X	18:40:50	L'ora.
%y	02	Anno nel secolo.
%Y	2002	Anno.
%Z	CEST	Nome della <i>timezone</i> .
%%	%	Il carattere %.

Tabella 6.15: Valori previsti dallo standard ANSI C per modificatore della stringa di formato di `strftime`.

modificatori di tab. 6.15 più altre estensioni, ma per i dettagli a questo riguardo si rimanda alla lettura della pagina di manuale.

Si tenga presente comunque che anche in caso di scansione completamente riuscita la funzione sovrascrive soltanto i campi di `tm` indicati dal formato, la struttura originaria infatti non viene inizializzata e gli altri campi restano ai valori che avevano in precedenza.

6.5 La gestione degli errori

In questa sezione esamineremo le caratteristiche principali della gestione degli errori in un sistema unix-like. Infatti a parte il caso particolare di alcuni segnali (che tratteremo in cap. 7) in un sistema unix-like il kernel non avvisa mai direttamente un processo dell'occorrenza di un errore nell'esecuzione di una funzione, ma di norma questo viene riportato semplicemente usando un opportuno valore di ritorno della funzione invocata. Inoltre il sistema di classificazione degli errori è stato progettato sull'architettura a processi, e presenta una serie di problemi nel caso lo si debba usare con i *thread*.

6.5.1 La variabile `errno`

Quasi tutte le funzioni delle librerie del C sono in grado di individuare e riportare condizioni di errore, ed è una norma fondamentale di buona programmazione controllare *sempre* che le funzioni chiamate si siano concluse correttamente.

In genere le funzioni di libreria usano un valore speciale per indicare che c'è stato un errore. Di solito questo valore, a seconda della funzione, è `-1` o un puntatore nullo o la costante `EOF`; ma questo valore segnala solo che c'è stato un errore, e non il tipo di errore.

Per riportare il tipo di errore il sistema usa la variabile globale `errno`, definita nell'header `errno.h`. Come accennato l'uso di una variabile globale può comportare problemi nel caso dei

thread, ma lo standard ISO C consente anche di definire `errno` come un cosiddetto “*modifiable lvalue*”, cosa che consente di usare anche una macro, e questo è infatti il metodo usato da Linux per renderla locale ai singoli *thread*.

La variabile è in genere definita come `volatile` dato che può essere cambiata in modo asincrono da un segnale; per un esempio si veda sez. 7.3.6 ricordando quanto trattato in sez. 3.4.2. Dato che un gestore di segnale scritto bene si cura di salvare e ripristinare il valore della variabile all'uscita, nella programmazione normale, quando si può fare l'assunzione che i gestori di segnali siano ben scritti, di questo non è necessario preoccuparsi.

I valori che può assumere `errno` sono riportati in app. A, nell'header `errno.h` sono anche definiti i nomi simbolici per le costanti numeriche che identificano i vari errori che abbiamo citato fin dall'inizio nelle descrizioni delle funzioni. Essi iniziano tutti per E e si possono considerare come nomi riservati, per questo abbiamo sempre fatto riferimento a questi nomi, e lo faremo più avanti quando descriveremo i possibili errori restituiti dalle funzioni. Il programma di esempio `errcode` stampa il codice relativo ad un valore numerico con l'opzione `-l`.

Il valore di `errno` viene sempre impostato a zero all'avvio di un programma, e la gran parte delle funzioni di libreria impostano `errno` ad un valore diverso da zero in caso di errore. Il valore è invece indefinito in caso di successo, perché anche se una funzione di libreria ha successo, potrebbe averne chiamate altre al suo interno che potrebbero essere fallite anche senza compromettere il risultato finale, modificando però `errno`.

Pertanto un valore non nullo di `errno` non è sintomo di errore (potrebbe essere il risultato di un errore precedente) e non lo si può usare per determinare quando o se una chiamata a funzione è fallita. La procedura corretta da seguire per identificare un errore è sempre quella di controllare `errno` immediatamente dopo aver verificato il fallimento della funzione attraverso il suo codice di ritorno.

6.5.2 Le funzioni `strerror` e `perror`

Benché gli errori siano identificati univocamente dal valore numerico di `errno` le librerie provvedono alcune funzioni e variabili utili per riportare in opportuni messaggi le condizioni di errore verificatesi. La prima funzione che si può usare per ricavare i messaggi di errore è `strerror`, il cui prototipo è:

```
#include <string.h>
char *strerror(int errnum)
```

Restituisce una stringa con un messaggio di errore.

La funzione ritorna il puntatore alla stringa con il messaggio di errore, `errno` non viene modificato.

La funzione ritorna il puntatore alla stringa contenente il messaggio di errore corrispondente al valore di `errnum`, se questo non è un valore valido verrà comunque restituita una stringa valida contenente un messaggio che dice che l'errore è sconosciuto nella forma. La versione della *glibc* non modifica il valore di `errno` in caso di errore, ma questo non è detto valga per altri sistemi in quanto lo standard POSIX.1-2001 permette che ciò avvenga. Non si faccia affidamento su questa caratteristica se si vogliono scrivere programmi portabili.

In generale `strerror` viene usata passando direttamente `errno` come argomento, ed il valore di quest'ultima non verrà modificato. La funzione inoltre tiene conto del valore della variabile di ambiente `LC_MESSAGES` per usare le appropriate traduzioni dei messaggi d'errore nella localizzazione presente.

La funzione `strerror` utilizza una stringa statica che non deve essere modificata dal programma; essa è utilizzabile solo fino ad una chiamata successiva a `strerror` o `perror` e nessun'altra funzione di libreria tocca questa stringa. In ogni caso l'uso di una stringa statica

rende la funzione non rientrante, per cui nel caso si usino i *thread* la *glibc* fornisce una apposita versione rientrante `strerror_r`, il cui prototipo è:

```
#include <string.h>
char * strerror_r(int errnum, char *buf, size_t size)
                                Restituisce una stringa con un messaggio di errore.
```

La funzione ritorna l'indirizzo del messaggio in caso di successo e NULL per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL si è specificato un valore di `errnum` non valido.

ERANGE la lunghezza di `buf` è insufficiente a contenere la stringa di errore.

Si tenga presente che questa è la versione prevista normalmente nella *glibc*, ed effettivamente definita in `string.h`, ne esiste una analoga nello standard SUSv3 (riportata anche nella pagina di manuale), che restituisce `int` al posto di `char *`, e che tronca la stringa restituita a `size`, a cui si accede definendo le opportune macro (per le quali si rimanda alla lettura della pagina di manuale).

La funzione è analoga a `strerror` ma restituisce la stringa di errore nel buffer `buf` che il singolo *thread* deve allocare autonomamente per evitare i problemi connessi alla condivisione del buffer statico. Il messaggio è copiato fino alla dimensione massima del buffer, specificata dall'argomento `size`, che deve comprendere pure il carattere di terminazione; altrimenti la stringa risulterà troncata.

Una seconda funzione usata per riportare i codici di errore in maniera automatizzata sullo standard error è `perror`, il cui prototipo è:

```
#include <stdio.h>
void perror(const char *message)
                                Stampa un messaggio di errore personalizzato.
```

La funzione non ritorna nulla e non modifica `errno`.

I messaggi di errore stampati sono gli stessi di `strerror`, (riportati in app. A), e, usando il valore corrente di `errno`, si riferiscono all'ultimo errore avvenuto. La stringa specificata con `message` viene stampata prima del messaggio d'errore, consentono una personalizzazione (ad esempio l'indicazione del contesto in cui si è verificato), seguita dai due punti e da uno spazio, il messaggio è terminato con un a capo. Il messaggio può essere riportato anche usando le due variabili globali:

```
const char *sys_errlist[];
int sys_nerr;
```

dichiarate in `errno.h`. La prima contiene i puntatori alle stringhe di errore indicizzati da `errno`; la seconda esprime il valore più alto per un codice di errore, l'utilizzo di una di queste stringhe è sostanzialmente equivalente a quello di `strerror`, ma dato che non è detto che `sys_errlist` sia stato aggiornato in caso di aggiunta di nuovi errori, il suo uso è deprecato e si deve sempre usare `perror`.

In fig. 6.12 è riportata la sezione attinente del codice del programma `errcode`, che può essere usato per stampare i messaggi di errore e le costanti usate per identificare i singoli errori. Il sorgente completo del programma è allegato nel file `ErrCode.c` e contiene pure la gestione delle opzioni e tutte le definizioni necessarie ad associare il valore numerico alla costante simbolica. In particolare si è riportata la sezione che converte la stringa passata come argomento in un intero (1-2), controllando con i valori di ritorno di `strtol` che la conversione sia avvenuta correttamente (4-10), e poi stampa, a seconda dell'opzione scelta il messaggio di errore (11-14) o la macro (15-17) associate a quel codice.

```

1  /* convert string to number */
2  err = strtol(argv[optind], NULL, 10);
3  /* testing error condition on conversion */
4  if (err==LONG_MIN) {
5      perror("Underflow_on_error_code");
6      return 1;
7  } else if (err==LONG_MAX) {
8      perror("Overflow_on_error_code");
9      return 1;
10 }
11 /* conversion is fine */
12 if (message) {
13     printf("Error_message_for_%d_is_%s\n", err, strerror(err));
14 }
15 if (label) {
16     printf("Error_label_for_%d_is_%s\n", err, err_code[err]);
17 }

```

Figura 6.12: Codice per la stampa del messaggio di errore standard.

6.5.3 Alcune estensioni GNU

Le precedenti funzioni sono quelle definite ed usate nei vari standard; la *glibc* ha però introdotto una serie di estensioni “GNU” che forniscono alcune funzionalità aggiuntive per una gestione degli errori semplificata e più efficiente.

La prima estensione consiste in due variabili, `char * program_invocation_name` e `char * program_invocation_short_name` che consentono di ricavare il nome del proprio programma. Queste sono utili quando si deve aggiungere il nome del programma al messaggio d’errore, cosa comune quando si ha un programma che non viene lanciato da linea di comando e salva gli errori in un file di log. La prima contiene il nome usato per lanciare il programma dalla shell ed in sostanza è equivalente ad `argv[0]`; la seconda mantiene solo il nome del programma eliminando eventuali directory qualora questo sia stato lanciato con un *pathname*.

Una seconda estensione cerca di risolvere uno dei problemi che si hanno con l’uso di `perror`, dovuto al fatto che non c’è flessibilità su quello che si può aggiungere al messaggio di errore, che può essere solo una stringa. In molte occasioni invece serve poter scrivere dei messaggi con maggiori informazioni. Ad esempio negli standard di programmazione GNU si richiede che ogni messaggio di errore sia preceduto dal nome del programma, ed in generale si può voler stampare il contenuto di qualche variabile per facilitare la comprensione di un eventuale problema. Per questo la *glibc* definisce la funzione `error`, il cui prototipo è:

<pre>#include <stdio.h> void error(int status, int errnum, const char *format, ...)</pre>
Stampa un messaggio di errore formattato.

La funzione non ritorna nulla e non riporta errori.

La funzione fa parte delle estensioni GNU per la gestione degli errori, l’argomento `format` segue la stessa sintassi di `printf` (vedi sez. 5.3.6), ed i relativi argomenti devono essere forniti allo stesso modo, mentre `errnum` indica l’errore che si vuole segnalare (non viene quindi usato il valore corrente di `errno`).

La funzione stampa sullo *standard error* il nome del programma, come indicato dalla variabile globale `program_name`, seguito da due punti ed uno spazio, poi dalla stringa generata

da `format` e dagli argomenti seguenti, seguita da due punti ed uno spazio infine il messaggio di errore relativo ad `errno`, il tutto è terminato da un a capo.

Il comportamento della funzione può essere ulteriormente controllato se si definisce una variabile `error_print_progname` come puntatore ad una funzione `void` che restituisce `void` che si incarichi di stampare il nome del programma.

L'argomento `status` può essere usato per terminare direttamente il programma in caso di errore, nel qual caso `error` dopo la stampa del messaggio di errore chiama `exit` con questo stato di uscita. Se invece il valore è nullo `error` ritorna normalmente ma viene incrementata un'altra variabile globale, `error_message_count`, che tiene conto di quanti errori ci sono stati.

Un'altra funzione per la stampa degli errori, ancora più sofisticata, che prende due argomenti aggiuntivi per indicare linea e file su cui è avvenuto l'errore è `error_at_line`; il suo prototipo è:

```
#include <stdio.h>
void error_at_line(int status, int errno, const char *fname, unsigned int lineno,
                  const char *format, ...)
    Stampa un messaggio di errore formattato.
```

La funzione non ritorna nulla e non riporta errori.

ed il suo comportamento è identico a quello di `error` se non per il fatto che, separati con il solito due punti-spazio, vengono inseriti un nome di file indicato da `fname` ed un numero di linea subito dopo la stampa del nome del programma. Inoltre essa usa un'altra variabile globale, `error_one_per_line`, che impostata ad un valore diverso da zero fa sì che errori relativi alla stessa linea non vengano ripetuti.

Capitolo 7

I segnali

I segnali sono il primo e più semplice meccanismo di comunicazione nei confronti dei processi. Nella loro versione originale essi portano con sé nessuna informazione che non sia il loro tipo; si tratta in sostanza di un'interruzione software portata ad un processo.

In genere essi vengono usati dal kernel per riportare ai processi situazioni eccezionali (come errori di accesso, eccezioni aritmetiche, ecc.) ma possono anche essere usati come forma elementare di comunicazione fra processi (ad esempio vengono usati per il controllo di sessione), per notificare eventi (come la terminazione di un processo figlio), ecc.

In questo capitolo esamineremo i vari aspetti della gestione dei segnali, partendo da una introduzione relativa ai concetti base con cui essi vengono realizzati, per poi affrontarne la classificazione a secondo di uso e modalità di generazione fino ad esaminare in dettaglio le funzioni e le metodologie di gestione avanzate e le estensioni fatte all'interfaccia classica nelle nuove versioni dello standard POSIX.

7.1 Introduzione

In questa sezione esamineremo i concetti generali relativi ai segnali, vedremo le loro caratteristiche di base, introdurremo le nozioni di fondo relative all'architettura del funzionamento dei segnali e alle modalità con cui il sistema gestisce l'interazione fra di essi ed i processi.

7.1.1 I concetti base

Come il nome stesso indica i segnali sono usati per notificare ad un processo l'occorrenza di un qualche evento. Gli eventi che possono generare un segnale sono vari; un breve elenco di possibili cause per l'emissione di un segnale è il seguente:

- un errore del programma, come una divisione per zero o un tentativo di accesso alla memoria fuori dai limiti validi;
- la terminazione di un processo figlio;
- la scadenza di un timer o di un allarme;
- il tentativo di effettuare un'operazione di input/output che non può essere eseguita;
- una richiesta dell'utente dal terminale di terminare o fermare il programma.
- l'invio esplicito da parte del processo stesso o di un altro.

Ciascuno di questi eventi, compresi gli ultimi due che pure sono controllati dall'utente o da un altro processo, comporta l'intervento diretto da parte del kernel che causa la generazione di un particolare tipo di segnale.

Quando un processo riceve un segnale, invece del normale corso del programma, viene eseguita una azione predefinita o una apposita funzione di gestione che può essere stata specificata dall'utente, nel qual caso si dice che si *intercetta* il segnale. Riprendendo la terminologia originale da qui in avanti faremo riferimento a questa funzione come al *gestore* del segnale, traduzione approssimata dell'inglese *signal handler*.

7.1.2 Le *semantiche* del funzionamento dei segnali

Negli anni il comportamento del sistema in risposta ai segnali è stato modificato in vari modi nelle differenti implementazioni di Unix. Si possono individuare due tipologie fondamentali di comportamento dei segnali (dette *semantiche*) che vengono chiamate rispettivamente *semantica affidabile* (o *reliable*) e *semantica inaffidabile* (o *unreliable*).

Nella *semantica inaffidabile*, che veniva implementata dalle prime versioni di Unix, la funzione di gestione del segnale specificata dall'utente non restava attiva una volta che era stata eseguita; era perciò compito dell'utente ripetere l'installazione dello stesso all'interno del *gestore* del segnale in tutti quei casi in cui si voleva che esso restasse attivo.

```
1 int sig_handler();                /* handler function */
2 int main()
3 {
4     ...
5     signal(SIGINT, sig_handler); /* establish handler */
6     ...
7 }
8
9 int sig_handler()
10 {
11     signal(SIGINT, sig_handler); /* reestablish handler */
12     ...                          /* process signal */
13 }
```

Figura 7.1: Esempio di codice di un gestore di segnale per la semantica inaffidabile.

In questo caso però è possibile una situazione in cui i segnali possono essere perduti. Si consideri il segmento di codice riportato in fig. 7.1: nel programma principale viene installato un gestore (5), la cui prima operazione (11) è quella di reinstallare se stesso. Se nell'esecuzione del gestore fosse arrivato un secondo segnale prima che esso abbia potuto eseguire la reinstallazione di se stesso per questo secondo segnale verrebbe eseguito il comportamento predefinito, il che può comportare, a seconda dei casi, la perdita del segnale (se l'impostazione predefinita è quella di ignorarlo) o la terminazione immediata del processo; in entrambi i casi l'azione prevista dal gestore non verrebbe eseguita.

Questa è la ragione per cui l'implementazione dei segnali secondo questa semantica viene chiamata *inaffidabile*: infatti la ricezione del segnale e la reinstallazione del suo gestore non sono operazioni atomiche, e sono sempre possibili delle *race condition* (si ricordi sez. 3.4). Un altro problema è che in questa semantica non esiste un modo per bloccare i segnali quando non si vuole che arrivino; i processi possono ignorare il segnale, ma non è possibile istruire il

sistema a non fare nulla in occasione di un segnale, pur mantenendo memoria del fatto che è avvenuto.

Nella semantica *affidabile* (quella utilizzata da Linux e da ogni Unix moderno) il gestore una volta installato resta attivo e non si hanno tutti i problemi precedenti. In questa semantica i segnali vengono *generati* dal kernel per un processo all'occorrenza dell'evento che causa il segnale. In genere questo viene fatto dal kernel impostando un apposito campo della `task_struct` del processo nella *process table* (si veda fig. 3.2).

Si dice che il segnale viene *consegnato* al processo (dall'inglese *delivered*) quando viene eseguita l'azione per esso prevista, mentre per tutto il tempo che passa fra la generazione del segnale e la sua consegna esso è detto *pendente* (o *pending*). In genere questa procedura viene effettuata dallo *scheduler* quando, riprendendo l'esecuzione del processo in questione, verifica la presenza del segnale nella `task_struct` e mette in esecuzione il gestore.

In questa semantica un processo ha la possibilità di bloccare la consegna dei segnali, in questo caso, se l'azione per il suddetto segnale non è quella di ignorarlo, il segnale resta *pendente* fintanto che il processo non lo sblocca (nel qual caso viene consegnato) o imposta l'azione corrispondente per ignorarlo.

Si tenga presente che il kernel stabilisce cosa fare con un segnale che è stato bloccato al momento della consegna, non quando viene generato; questo consente di cambiare l'azione per il segnale prima che esso venga consegnato, e si può usare la funzione `sigpending` (vedi sez. 7.4.4) per determinare quali segnali sono bloccati e quali sono pendenti.

Infine occorre precisare che i segnali predatano il supporto per i *thread* e vengono sempre inviati al processo come insieme, cosa che può creare incertezza nel caso questo sia *multi-thread*. In tal caso quando è possibile determinare quale è il *thread* specifico che deve ricevere il segnale, come avviene per i segnali di errore, questo sarà inviato solo a lui, altrimenti sarà inviato a discrezione del kernel ad uno qualunque dei *thread* del processo che possa riceverlo (che cioè non blocchi il segnale), torneremo sull'argomento in sez. 12.4.3.

7.1.3 Tipi di segnali

In generale si tende a classificare gli eventi che possono generare dei segnali in tre categorie principali: errori, eventi esterni e richieste esplicite.

Un errore significa che un programma ha fatto qualcosa di sbagliato e non può continuare ad essere eseguito. Non tutti gli errori causano dei segnali, in genere le condizioni di errore più comuni comportano la restituzione di un codice di errore da parte di una funzione di libreria. Sono gli errori che possono avvenire nell'esecuzione delle istruzioni di un programma, come le divisioni per zero o l'uso di indirizzi di memoria non validi, che causano l'emissione di un segnale.

Un evento esterno ha in genere a che fare con le operazioni di lettura e scrittura su file, o con l'interazione con dispositivi o con altri processi; esempi di segnali di questo tipo sono quelli legati all'arrivo di dati in ingresso, scadenze di un timer, terminazione di processi figli, la pressione dei tasti di stop o di suspend su un terminale.

Una richiesta esplicita significa l'uso da parte di un programma delle apposite funzioni di sistema, come `kill` ed affini (vedi sez. 7.3.3) per la generazione “manuale” di un segnale.

Si dice poi che i segnali possono essere *asincroni* o *sincroni*. Un segnale *sincrono* è legato ad una azione specifica di un programma ed è inviato (a meno che non sia bloccato) durante tale azione. Molti errori generano segnali *sincroni*, così come la richiesta esplicita da parte del processo tramite le chiamate al sistema. Alcuni errori come la divisione per zero non sono completamente sincroni e possono arrivare dopo qualche istruzione.

I segnali *asincroni* sono generati da eventi fuori dal controllo del processo che li riceve, e arrivano in tempi imprevedibili nel corso dell'esecuzione del programma. Eventi esterni come la terminazione di un processo figlio generano segnali *asincroni*, così come le richieste di generazione di un segnale effettuate da altri processi.

In generale un tipo di segnale o è sincrono o è asincrono, salvo il caso in cui esso sia generato attraverso una richiesta esplicita tramite chiamata al sistema, nel qual caso qualunque tipo di segnale (quello scelto nella chiamata) può diventare sincrono o asincrono a seconda che sia generato internamente o esternamente al processo.

7.1.4 La notifica dei segnali

Come accennato quando un segnale viene generato, se la sua azione predefinita non è quella di essere ignorato, il kernel prende nota del fatto nella `task_struct` del processo; si dice così che il segnale diventa *pendente* (o *pending*), e rimane tale fino al momento in cui verrà notificato al processo o verrà specificata come azione quella di ignorarlo.

Normalmente l'invio al processo che deve ricevere il segnale è immediato ed avviene non appena questo viene rimesso in esecuzione dallo *scheduler* che esegue l'azione specificata. Questo a meno che il segnale in questione non sia stato bloccato prima della notifica, nel qual caso l'invio non avviene ed il segnale resta *pendente* indefinitamente.

Quando lo si sblocca un segnale *pendente* sarà subito notificato. Si tenga presente però che tradizionalmente i segnali *pendenti* non si accodano, alla generazione infatti il kernel marca un flag nella `task_struct` del processo, per cui se prima della notifica ne vengono generati altri il flag è comunque marcato, ed il gestore viene eseguito sempre una sola volta. In realtà questo non vale nel caso dei cosiddetti segnali *real-time*, che vedremo in sez. 7.5.1, ma questa è una funzionalità avanzata che per ora tralascieremo.

Si ricordi inoltre che se l'azione specificata per un segnale è quella di essere ignorato questo sarà scartato immediatamente al momento della sua generazione, e questo anche se in quel momento il segnale è bloccato, perché bloccare su un segnale significa bloccarne la notifica. Per questo motivo un segnale, fintanto che viene ignorato, non sarà mai notificato, anche se prima è stato bloccato ed in seguito si è specificata una azione diversa, nel qual caso solo i segnali successivi alla nuova specificazione saranno notificati.

Una volta che un segnale viene notificato, che questo avvenga subito o dopo una attesa più o meno lunga, viene eseguita l'azione specificata per il segnale. Per alcuni segnali (per la precisione SIGKILL e SIGSTOP) questa azione è predeterminata dal kernel e non può essere mai modificata, ma per tutti gli altri si può selezionare una delle tre possibilità seguenti:

- ignorare il segnale;
- intercettare il segnale, ed utilizzare il gestore specificato;
- accettare l'azione predefinita per quel segnale.

Un programma può specificare queste scelte usando le due funzioni `signal` e `sigaction`, che tratteremo rispettivamente in sez. 7.3.2 e sez. 7.4.3. Se si è installato un gestore sarà quest'ultimo ad essere eseguito alla notifica del segnale. Inoltre il sistema farà sì che mentre viene eseguito il gestore di un segnale, quest'ultimo venga automaticamente bloccato, così si possono evitare alla radice possibili *race condition*.

Nel caso non sia stata specificata un'azione, viene utilizzata la cosiddetta azione predefinita che, come vedremo in sez. 7.2.1, è propria di ciascun segnale. Nella maggior parte dei casi questa azione comporta la terminazione immediata del processo, ma per alcuni segnali che rappresentano eventi innocui l'azione predefinita è di essere ignorati. Inoltre esistono alcuni

segnali la cui azione è semplicemente quella di fermare l'esecuzione del programma, vale a dire portarlo nello stato di *stopped* (lo stato T, si ricordi tab. 3.8 e quanto illustrato in sez. 3.3.1).

Quando un segnale termina un processo il padre può determinare la causa della terminazione esaminandone lo stato di uscita così come viene riportato dalle funzioni `wait` e `waitpid` (vedi sez. 3.1.5). Questo ad esempio è il modo in cui la shell determina i motivi della terminazione di un programma e scrive un eventuale messaggio di errore.

I segnali che rappresentano errori del programma (divisione per zero o violazioni di accesso) hanno come ulteriore caratteristica della loro azione predefinita, oltre a terminare il processo, quella di scrivere nella directory di lavoro corrente del processo di un file `core` su cui viene salvata l'immagine della memoria del processo.

Questo file costituisce il cosiddetto *core dump*, e contenendo l'immagine della memoria del processo, consente di risalire allo stato dello *stack* (vedi sez. 2.2.2) prima della terminazione. Questo permette di esaminare il contenuto del file un secondo tempo con un apposito programma (un *debugger* come `gdb`) per investigare sulla causa dell'errore, ed in particolare, grazie appunto ai dati dello *stack*, consente di identificare quale funzione ha causato l'errore.

Si tenga presente che il *core dump* viene creato non solo in caso di errore effettivo, ma anche se il segnale per cui la sua creazione è prevista nell'azione dell'azione predefinita viene inviato al programma con una delle funzioni `kill`, `raise`, ecc.

7.2 La classificazione dei segnali

Esamineremo in questa sezione quali sono i vari segnali definiti nel sistema, quali sono le loro caratteristiche e la loro tipologia, tratteremo le varie macro e costanti che permettono di identificarli, e illustreremo le funzioni che ne stampano la descrizione.

7.2.1 I segnali standard

Ciascun segnale è identificato dal kernel con un numero, ma benché per alcuni segnali questi numeri siano sempre gli stessi, tanto da essere usati come sinonimi, l'uso diretto degli identificativi numerici da parte dei programmi è comunque da evitare, in quanto essi non sono mai stati standardizzati e possono variare a seconda dell'implementazione del sistema, e nel caso di Linux anche a seconda della architettura hardware e della versione del kernel.

Quelli che invece sono stati, almeno a grandi linee, standardizzati, sono i nomi dei segnali e le costanti di preprocessore che li identificano, che sono tutte nella forma `SIGnome`, e sono queste che devono essere usate nei programmi. Come tutti gli altri nomi e le funzioni che concernono i segnali, esse sono definite nell'header di sistema `signal.h`.

In tab. 7.1 si è riportato l'elenco completo dei segnali ordinari definiti su Linux per tutte le possibili architetture (tratteremo quelli *real-time* in sez. 7.5.1). Ma si tenga presente che solo quelli elencati nella prima sezione della tabella sono presenti su tutte le architetture. Nelle sezioni successive si sono riportati rispettivamente quelli che esistono solo sull'architettura PC e quelli che non esistono sull'architettura PC, ma sono definiti su altre.

Alcuni segnali erano previsti fin dallo standard ANSI C, ed i segnali sono presenti in tutti i sistemi unix-like, ma l'elenco di quelli disponibili non è uniforme, ed alcuni di essi sono presenti solo su alcune implementazioni o architetture hardware, ed anche il loro significato può variare. Per questo si sono riportati nella seconda colonna della tabella riporta gli standard in cui ciascun segnale è stato definito, indicati con altrettante lettere da interpretare secondo la legenda di tab. 7.2. Si tenga presente che il significato dei segnali è abbastanza indipendente dalle implementazioni solo per quelli definiti negli standard POSIX.1-1990 e POSIX.1-2001.

Segnale	Standard	Azione	Descrizione
SIGHUP	P	T	Hangup o terminazione del processo di controllo.
SIGINT	PA	T	Interrupt da tastiera (C-c).
SIGQUIT	P	C	Quit da tastiera (C-y).
SIGILL	PA	C	Istruzione illecita.
SIGTRAP	S	C	Trappole per un Trace/breakpoint.
SIGABRT	PA	C	Segnale di abort da abort.
SIGIOT	B	C	Trappola di I/O. Sinonimo di SIGABRT.
SIGBUS	BS	C	Errore sul bus (bad memory access).
SIGFPE	AP	C	Errore aritmetico.
SIGKILL	P	T	Segnale di terminazione forzata.
SIGUSR1	P	T	Segnale utente numero 1.
SIGSEGV	AP	C	Errore di accesso in memoria.
SIGUSR2	P	T	Segnale utente numero 2.
SIGPIPE	P	T	<i>Pipe</i> spezzata.
SIGALRM	P	T	Segnale del timer da alarm.
SIGTERM	AP	T	Segnale di terminazione (C-^).
SIGCHLD	P	I	Figlio terminato o fermato.
SIGCONT	P	–	Continua se fermato.
SIGSTOP	P	S	Ferma il processo.
SIGTSTP	P	S	Pressione del tasto di stop sul terminale.
SIGTTIN	P	S	Input sul terminale per un processo in background.
SIGTTOU	P	S	Output sul terminale per un processo in background.
SIGURG	BS	I	Ricezione di una <i>urgent condition</i> su un socket.
SIGXCPU	BS	C	Ecceduto il limite sul tempo di CPU.
SIGXFSZ	BS	C	Ecceduto il limite sulla dimensione dei file.
SIGVTALRM	BS	T	Timer di esecuzione scaduto.
SIGPROF	BS	T	Timer del <i>profiling</i> scaduto.
SIGWINCH	B	I	Finestra ridimensionata (4.3BSD, Sun).
SIGIO	B	T	L'I/O è possibile.
SIGPOLL	VS	T	<i>Pollable event</i> , sinonimo di SIGIO.
SIGPWR	V	T	Fallimento dell'alimentazione.
SIGSYS	VS	C	<i>system call</i> sbagliata.
SIGSTKFLT	?	T	Errore sullo stack del coprocessore (inusato).
SIGUNUSED	?	C	Segnale inutilizzato (sinonimo di SIGSYS).
SIGCLD	V	I	Sinonimo di SIGCHLD.
SIGEMT	V	C	Trappola di emulatore.
SIGINFO	B	T	Sinonimo di SIGPWR.
SIGLOST	?	T	Perso un lock sul file, sinonimo di SIGIO (inusato).

Tabella 7.1: Lista dei segnali ordinari in Linux.

Come accennato in sez. 7.1.4 a ciascun segnale è associata una specifica azione predefinita che viene eseguita quando nessun gestore è installato. Le azioni predefinite possibili, che abbiamo già descritto in sez. 7.1.4, sono state riportate in tab. 7.1 nella terza colonna, e di nuovo sono state indicate con delle lettere la cui legenda completa è illustrata in tab. 7.3).

Sigla	Standard
P	POSIX.1-1990.
B	BSD (4.2 BSD e Sun).
A	ANSI C.
S	SUSv2 (e POSIX.1-2001).
V	System V.
?	Ignoto.

Tabella 7.2: Legenda dei valori degli standard riportati nella seconda colonna di tab. 7.1.

Si inoltre noti come SIGCONT sia l'unico segnale a non avere l'indicazione di una azione

predefinita nella terza colonna di tab. 7.1, questo perché il suo effetto è sempre quello di far ripartire un programma in stato T fermato da un segnale di stop. Inoltre i segnali SIGSTOP e SIGKILL si distinguono da tutti gli altri per la specifica caratteristica di non potere essere né intercettati, né bloccati, né ignorati.

Sigla	Significato
T	L'azione predefinita è terminare il processo.
C	L'azione predefinita è terminare il processo e scrivere un <i>core dump</i> .
I	L'azione predefinita è ignorare il segnale.
S	L'azione predefinita è fermare il processo.

Tabella 7.3: Legenda delle azioni predefinite dei segnali riportate nella terza colonna di tab. 7.1.

Il numero totale di segnali presenti è dato dalla macro NSIG (e tiene conto anche di quelli *real-time*) e dato che i numeri dei segnali sono allocati progressivamente, essa corrisponde anche al successivo del valore numerico assegnato all'ultimo segnale definito. La descrizione dettagliata del significato dei precedenti segnali, raggruppati per tipologia, verrà affrontata nei paragrafi successivi.

7.2.2 I segnali di errore

Questi segnali sono generati quando il sistema, o in certi casi direttamente l'hardware (come per i *page fault* non validi o le eccezioni del processore) rileva un qualche errore insanabile nel programma in esecuzione. In generale la generazione di questi segnali significa che il programma ha dei gravi problemi (ad esempio ha dereferenziato un puntatore non valido o ha eseguito una operazione aritmetica proibita) e l'esecuzione non può essere proseguita.

In genere si intercettano questi segnali per permettere al programma di terminare in maniera pulita, ad esempio per ripristinare le impostazioni della console o eliminare i file di lock prima dell'uscita. In questo caso il gestore deve concludersi ripristinando l'azione predefinita e rialzando il segnale, in questo modo il programma si concluderà senza effetti spiacevoli, ma riportando lo stesso stato di uscita che avrebbe avuto se il gestore non ci fosse stato.

L'azione predefinita per tutti questi segnali è causare la terminazione del processo che li ha causati. In genere oltre a questo il segnale provoca pure la registrazione su disco di un file di *core dump*, che un debugger può usare per ricostruire lo stato del programma al momento della terminazione. Questi segnali sono:

- SIGFPE Riporta un errore aritmetico fatale. Benché il nome derivi da *floating point exception* si applica a tutti gli errori aritmetici compresa la divisione per zero e l'overflow. Se il gestore ritorna il comportamento del processo è indefinito, ed ignorare questo segnale può condurre ad un ciclo infinito.
- SIGILL Il nome deriva da *illegal instruction*, significa che il programma sta cercando di eseguire una istruzione privilegiata o inesistente, in generale del codice illecito. Poiché il compilatore del C genera del codice valido si ottiene questo segnale se il file eseguibile è corrotto o si stanno cercando di eseguire dei dati. Quest'ultimo caso può accadere quando si passa un puntatore sbagliato al posto di un puntatore a funzione, o si eccede la scrittura di un vettore di una variabile locale, andando a corrompere lo *stack*. Lo stesso segnale viene generato in caso di overflow dello *stack* o di problemi nell'esecuzione di un gestore. Se il gestore ritorna il comportamento del processo è indefinito.

SIGSEGV	Il nome deriva da <i>segment violation</i> , e significa che il programma sta cercando di leggere o scrivere in una zona di memoria protetta al di fuori di quella che gli è stata riservata dal sistema. In genere è il meccanismo della protezione della memoria che si accorge dell'errore ed il kernel genera il segnale. È tipico ottenere questo segnale dereferenziando un puntatore nullo o non inizializzato leggendo al di là della fine di un vettore. Se il gestore ritorna il comportamento del processo è indefinito.
SIGBUS	Il nome deriva da <i>bus error</i> . Come SIGSEGV questo è un segnale che viene generato di solito quando si dereferenzia un puntatore non inizializzato, la differenza è che SIGSEGV indica un accesso non permesso su un indirizzo esistente (al di fuori dallo <i>heap</i> o dallo <i>stack</i>), mentre SIGBUS indica l'accesso ad un indirizzo non valido, come nel caso di un puntatore non allineato.
SIGABRT	Il nome deriva da <i>abort</i> . Il segnale indica che il programma stesso ha rilevato un errore che viene riportato chiamando la funzione <code>abort</code> , che genera questo segnale.
SIGTRAP	È il segnale generato da un'istruzione di breakpoint o dall'attivazione del tracciamento per il processo. È usato dai programmi per il debugging e un programma normale non dovrebbe ricevere questo segnale.
SIGSYS	Sta ad indicare che si è eseguita una istruzione che richiede l'esecuzione di una <i>system call</i> , ma si è fornito un codice sbagliato per quest'ultima.
SIGEMT	Il nome sta per <i>emulation trap</i> . Il segnale non è previsto da nessuno standard ed è definito solo su alcune architetture che come il vecchio PDP11 prevedono questo tipo di interruzione, non è presente sui normali PC.

7.2.3 I segnali di terminazione

Questo tipo di segnali sono usati per terminare un processo; hanno vari nomi a causa del differente uso che se ne può fare, ed i programmi possono trattarli in maniera differente.

La ragione per cui può essere necessario intercettare questi segnali è che il programma può dover eseguire una serie di azioni di pulizia prima di terminare, come salvare informazioni sullo stato in cui si trova, cancellare file temporanei, o ripristinare delle condizioni alterate durante il funzionamento (come il modo del terminale o le impostazioni di una qualche periferica). L'azione predefinita di questi segnali è di terminare il processo, questi segnali sono:

SIGTERM	Il nome sta per <i>terminate</i> . È un segnale generico usato per causare la conclusione di un programma. È quello che viene generato di default dal comando <code>kill</code> . Al contrario di SIGKILL può essere intercettato, ignorato, bloccato. In genere lo si usa per chiedere in maniera “educata” ad un processo di concludersi.
SIGINT	Il nome sta per <i>interrupt</i> . È il segnale di interruzione per il programma. È quello che viene generato di default dal dall'invio sul terminale del carattere di controllo “INTR”, <i>interrupt</i> appunto, che viene generato normalmente dalla sequenza C-c sulla tastiera.
SIGQUIT	È analogo a SIGINT con la differenza che è controllato da un altro carattere di controllo, “QUIT”, corrispondente alla sequenza C-\ sulla tastiera. A differenza del precedente l'azione predefinita, oltre alla terminazione del processo,

comporta anche la creazione di un *core dump*. In genere lo si può pensare come corrispondente ad una condizione di errore del programma rilevata dall'utente. Per questo motivo non è opportuno fare eseguire al gestore di questo segnale le operazioni di pulizia normalmente previste (tipo la cancellazione di file temporanei), dato che in certi casi esse possono eliminare informazioni utili nell'esame dei *core dump*.

SIGKILL Il nome è utilizzato per terminare in maniera immediata qualunque programma. Questo segnale non può essere né intercettato, né ignorato, né bloccato, per cui causa comunque la terminazione del processo. In genere esso viene generato solo per richiesta esplicita dell'utente dal comando (o tramite la funzione) `kill`. Dato che non lo si può intercettare è sempre meglio usarlo come ultima risorsa quando metodi meno brutali, come `SIGTERM` o `C-c` non funzionano.

Se un processo non risponde a nessun altro segnale `SIGKILL` ne causa sempre la terminazione (in effetti il fallimento della terminazione di un processo da parte di `SIGKILL` costituirebbe un malfunzionamento del kernel). Talvolta è il sistema stesso che può generare questo segnale quando per condizioni particolari il processo non può più essere eseguito neanche per eseguire un gestore.

SIGHUP Il nome sta per *hang-up*. Segnala che il terminale dell'utente si è disconnesso, ad esempio perché si è interrotta la rete. Viene usato anche per riportare la terminazione del processo di controllo di un terminale a tutti i processi della sessione (vedi sez. 8.1), in modo che essi possano disconnettersi dal relativo terminale. Viene inoltre usato in genere per segnalare ai programmi di servizio (i cosiddetti *demoni*, vedi sez. 8.1.5), che non hanno un terminale di controllo, la necessità di reinizializzarsi e rileggere il file (o i file) di configurazione.

7.2.4 I segnali di allarme

Questi segnali sono generati dalla scadenza di un timer (vedi sez. 7.3.4). Il loro comportamento predefinito è quello di causare la terminazione del programma, ma con questi segnali la scelta predefinita è irrilevante, in quanto il loro uso presuppone sempre la necessità di un gestore. Questi segnali sono:

SIGALRM Il nome sta per *alarm*. Segnala la scadenza di un timer misurato sul tempo reale o sull'orologio di sistema. È normalmente usato dalla funzione `alarm`.

SIVGTALRM Il nome sta per *virtual alarm*. È analogo al precedente ma segnala la scadenza di un timer sul tempo di CPU usato dal processo.

SIGPROF Il nome sta per *profiling*. Indica la scadenza di un timer che misura sia il tempo di CPU speso direttamente dal processo che quello che il sistema ha speso per conto di quest'ultimo. In genere viene usato dagli strumenti che servono a fare la profilazione dell'utilizzo del tempo di CPU da parte del processo.

7.2.5 I segnali di I/O asincrono

Questi segnali operano in congiunzione con le funzioni di I/O asincrono. Per questo occorre comunque usare `fcntl` per abilitare un file descriptor a generare questi segnali. L'azione predefinita è di essere ignorati. Questi segnali sono:

SIGIO	Questo segnale viene inviato quando un file descriptor è pronto per eseguire dell'input/output. In molti sistemi solo i socket e i terminali possono generare questo segnale, in Linux questo può essere usato anche per i file, posto che la chiamata a <code>fcntl</code> che lo attiva abbia avuto successo.
SIGURG	Questo segnale è inviato quando arrivano dei dati urgenti o <i>out-of-band</i> su di un socket; per maggiori dettagli al proposito si veda sez. 18.1.4.
SIGPOLL	Questo segnale è definito nella standard POSIX.1-2001, ed è equivalente a SIGIO che invece deriva da BSD. Su Linux è definito per compatibilità con i sistemi System V.

7.2.6 I segnali per il controllo di sessione

Questi sono i segnali usati dal controllo delle sessioni e dei processi, il loro uso è specializzato e viene trattato in maniera specifica nelle sezioni in cui si trattano gli argomenti relativi. Questi segnali sono:

SIGCHLD	Questo è il segnale mandato al processo padre quando un figlio termina o viene fermato. L'azione predefinita è di ignorare il segnale, la sua gestione è trattata in sez. 3.1.5.
SIGCLD	Per Linux questo è solo un segnale identico al precedente e definito come sinonimo. Il nome è obsoleto, deriva dalla definizione del segnale su System V, ed oggi deve essere evitato.
SIGCONT	<p>Il nome sta per <i>continue</i>. Il segnale viene usato per fare ripartire un programma precedentemente fermato da SIGSTOP. Questo segnale ha un comportamento speciale, e fa sempre ripartire il processo prima della sua consegna. Il comportamento predefinito è di fare solo questo; il segnale non può essere bloccato. Si può anche installare un gestore, ma il segnale provoca comunque il riavvio del processo.</p> <p>La maggior parte dei programmi non hanno necessità di intercettare il segnale, in quanto esso è completamente trasparente rispetto all'esecuzione che riparte senza che il programma noti niente. Si possono installare dei gestori per far sì che un programma produca una qualche azione speciale se viene fermato e riavviato, come per esempio riscrivere un prompt, o inviare un avviso.</p>
SIGSTOP	Il segnale ferma l'esecuzione di un processo, lo porta cioè nello stato <i>stopped</i> (vedi sez. 3.3.1). Il segnale non può essere né intercettato, né ignorato, né bloccato.
SIGTSTP	Il nome sta per <i>interactive stop</i> . Il segnale ferma il processo interattivamente, ed è generato dal carattere " <i>SUSP</i> ", prodotto dalla combinazione di tasti C-z, ed al contrario di SIGSTOP può essere intercettato e ignorato. In genere un programma installa un gestore per questo segnale quando vuole lasciare il sistema o il terminale in uno stato definito prima di fermarsi; se per esempio un programma ha disabilitato l'eco sul terminale può installare un gestore per riabilitarlo prima di fermarsi.
SIGTTIN	Un processo non può leggere dal terminale se esegue una sessione di lavoro in <i>background</i> . Quando un processo in <i>background</i> tenta di leggere da un terminale

viene inviato questo segnale a tutti i processi della sessione di lavoro. L'azione predefinita è di fermare il processo. L'argomento è trattato in sez. 8.1.1.

SIGTTOU Segnale analogo al precedente SIGTTIN, ma generato quando si tenta di scrivere sul terminale o modificarne uno dei modi con un processo in *background*. L'azione predefinita è di fermare il processo, l'argomento è trattato in sez. 8.1.1.

7.2.7 I segnali di operazioni errate

Questi segnali sono usati per riportare al programma errori generati da operazioni da lui eseguite; non indicano errori del programma quanto errori che impediscono il completamento dell'esecuzione dovute all'interazione con il resto del sistema. L'azione predefinita di questi segnali è normalmente quella di terminare il processo, questi segnali sono:

SIGPIPE Sta per *Broken pipe*. Se si usano delle *pipe*, (o delle FIFO o dei socket) è necessario, prima che un processo inizi a scrivere su una di esse, che un altro l'abbia aperta in lettura (si veda sez. 11.1.1). Se il processo in lettura non è partito o è terminato inavvertitamente alla scrittura sulla *pipe* il kernel genera questo segnale. Se il segnale è bloccato, intercettato o ignorato la chiamata che lo ha causato fallisce, restituendo l'errore EPIPE.

SIGXCPU Sta per *CPU time limit exceeded*. Questo segnale è generato quando un processo eccede il limite impostato per il tempo di CPU disponibile, vedi sez. 6.3.2. Fino al kernel 2.2 terminava semplicemente il processo, a partire dal kernel 2.4, seguendo le indicazioni dello standard POSIX.1-2001 viene anche generato un *core dump*.

SIGXFSZ Sta per *File size limit exceeded*. Questo segnale è generato quando un processo tenta di estendere un file oltre le dimensioni specificate dal limite impostato per le dimensioni massime di un file, vedi sez. 6.3.2. Fino al kernel 2.2 terminava semplicemente il processo, a partire dal kernel 2.4, seguendo le indicazioni dello standard POSIX.1-2001 viene anche generato un *core dump*.

SIGLOST Sta per *Resource lost*. Tradizionalmente è il segnale che viene generato quando si perde un advisory lock su un file su NFS perché il server NFS è stato riavviato. Il progetto GNU lo utilizza per indicare ad un client il crollo inaspettato di un server. In Linux è definito come sinonimo di SIGIO e non viene più usato.

7.2.8 Ulteriori segnali

Raccogliamo qui infine una serie di segnali che hanno scopi differenti non classificabili in maniera omogenea. Questi segnali sono:

SIGUSR1 Insieme a SIGUSR2 è un segnale a disposizione dell'utente che lo può usare per quello che vuole. Viene generato solo attraverso l'invocazione della funzione `kill`. Entrambi i segnali possono essere utili per implementare una comunicazione elementare fra processi diversi, o per eseguire a richiesta una operazione utilizzando un gestore. L'azione predefinita è di terminare il processo.

SIGUSR2 È il secondo segnale a disposizione degli utenti. Per il suo utilizzo vale esattamente quanto appena detto per SIGUSR1.

SIGWINCH	Il nome sta per <i>window (size) change</i> e viene generato in molti sistemi (GNU/Linux compreso) quando le dimensioni (in righe e colonne) di un terminale vengono cambiate. Viene usato da alcuni programmi testuali per riformattare l'uscita su schermo quando si cambia dimensione a quest'ultimo. L'azione predefinita è di essere ignorato.
SIGINFO	Il segnale indica una richiesta di informazioni. È usato con il controllo di sessione, causa la stampa di informazioni da parte del processo leader del gruppo associato al terminale di controllo, gli altri processi lo ignorano. Su Linux però viene utilizzato come sinonimo di SIGPWR e l'azione predefinita è di terminare il processo.
SIGPWR	Il segnale indica un cambio nello stato di alimentazione di un eventuale gruppo di continuità e viene usato principalmente per segnalare l'assenza ed il ritorno della corrente. Viene usato principalmente con <code>init</code> per attivare o fermare le procedure di spegnimento automatico all'esaurimento delle batterie. L'azione predefinita è di terminare il processo.
SIGSTKFLT	Indica un errore nello stack del coprocessore matematico, è definito solo per le architetture PC, ma è completamente inusato. L'azione predefinita è di terminare il processo.

7.2.9 Le funzioni `strsignal` e `psignal`

Per la descrizione dei segnali il sistema mette a disposizione due funzioni che stampano un messaggio di descrizione specificando il numero del segnale con una delle costanti di tab. 7.1. In genere si usano quando si vuole notificare all'utente il segnale ricevuto, ad esempio nel caso di terminazione di un processo figlio o di un gestore che gestisce più segnali.

La prima funzione, `strsignal`, è una estensione GNU fornita dalla *glibc*, ed è accessibile solo avendo definito la macro `_GNU_SOURCE`, il suo comportamento è analogo a quello della funzione `strerror` (si veda sez. 6.5.2) usata per notificare gli errori:

```
#include <string.h>
char *strsignal(int signum)
```

Ottiene la descrizione di un segnale.

La funzione ritorna puntatore ad una stringa che descrive il segnale, non sono previste condizioni di errore ed `errno` non viene modificata.

La funzione ritorna sempre il puntatore ad una stringa che contiene la descrizione del segnale indicato dall'argomento `signum`, se questo non indica un segnale valido viene restituito il puntatore ad una stringa che segnala che il valore indicato non è valido. Dato che la stringa è allocata staticamente non se ne deve modificare il contenuto, che resta valido solo fino alla successiva chiamata di `strsignal`. Nel caso si debba mantenere traccia del messaggio sarà necessario copiarlo.

La seconda funzione, `psignal`, deriva da BSD ed è analoga alla funzione `perror` descritta in sez. 6.5.2, il suo prototipo è:

```
#include <signal.h>
void psignal(int sig, const char *s)
```

Stampa un messaggio di descrizione di un segnale.

La funzione non ritorna nulla e non prevede errori.

La funzione stampa sullo *standard error* un messaggio costituito dalla stringa passata nell'argomento `s`, seguita dal carattere di due punti ed una descrizione del segnale indicato dall'argomento `sig`.

Una modalità alternativa per utilizzare le descrizioni restituite da `strsignal` e `psignal` è quello di usare la variabile globale `sys_siglist`, che è definita in `signal.h` e può essere acceduta con la dichiarazione:

```
extern const char *const sys_siglist[];
```

L'array `sys_siglist` contiene i puntatori alle stringhe di descrizione, indicizzate per numero di segnale, per cui una chiamata del tipo di `char *decr = strsignal(SIGINT)` può essere sostituita dall'equivalente `char *decr = sys_siglist[SIGINT]`.

7.3 La gestione di base dei segnali

I segnali sono il primo e più classico esempio di eventi asincroni, cioè di eventi che possono accadere in un qualunque momento durante l'esecuzione di un programma. Per questa loro caratteristica la loro gestione non può essere effettuata all'interno del normale flusso di esecuzione dello stesso, ma è delegata appunto agli eventuali gestori che si sono installati.

In questa sezione vedremo come si effettua la gestione dei segnali, a partire dalla loro interazione con le *system call*, passando per le varie funzioni che permettono di installare i gestori e controllare le reazioni di un processo alla loro occorrenza.

7.3.1 Il comportamento generale del sistema

Abbiamo già trattato in sez. 7.1 le modalità con cui il sistema gestisce l'interazione fra segnali e processi, ci resta da esaminare però il comportamento delle *system call*; in particolare due di esse, `fork` ed `exec`, dovranno essere prese esplicitamente in considerazione, data la loro stretta relazione con la creazione di nuovi processi.

Come accennato in sez. 3.1.3 quando viene creato un nuovo processo esso eredita dal padre sia le azioni che sono state impostate per i singoli segnali, che la maschera dei segnali bloccati (vedi sez. 7.4.4). Invece tutti i segnali pendenti e gli allarmi vengono cancellati; essi infatti devono essere recapitati solo al padre, al figlio dovranno arrivare solo i segnali dovuti alle sue azioni.

Quando si mette in esecuzione un nuovo programma con `exec` (si ricordi quanto detto in sez. 3.1.6) tutti i segnali per i quali è stato installato un gestore vengono reimpostati a `SIG_DFL`. Non ha più senso infatti fare riferimento a funzioni definite nel programma originario, che non sono presenti nello spazio di indirizzi del nuovo programma.

Si noti che questo vale solo per le azioni per le quali è stato installato un gestore, viene mantenuto invece ogni eventuale impostazione dell'azione a `SIG_IGN`. Questo permette ad esempio alla shell di impostare ad `SIG_IGN` le risposte per `SIGINT` e `SIGQUIT` per i programmi eseguiti in background, che altrimenti sarebbero interrotti da una successiva pressione di C-c o C-y.

Per quanto riguarda il comportamento di tutte le altre *system call* si danno sostanzialmente due casi, a seconda che esse siano *lente* (*slow*) o *veloci* (*fast*). La gran parte di esse appartiene a quest'ultima categoria, che non è influenzata dall'arrivo di un segnale. Esse sono dette *veloci* in quanto la loro esecuzione è sostanzialmente immediata. La risposta al segnale viene sempre data dopo che la *system call* è stata completata, in quanto attendere per eseguire un gestore non comporta nessun inconveniente.

In alcuni casi però alcune *system call* possono bloccarsi indefinitamente e per questo motivo vengono chiamate *lente* o *bloccanti*. In questo caso non si può attendere la conclusione della *system call*, perché questo renderebbe impossibile una risposta pronta al segnale, per cui il gestore viene eseguito prima che la *system call* sia ritornata. Un elenco dei casi in cui si presenta questa situazione è il seguente:

- la lettura da file che possono bloccarsi in attesa di dati non ancora presenti (come per certi file di dispositivo, i socket o le *pipe*);
- la scrittura sugli stessi file, nel caso in cui dati non possano essere accettati immediatamente (di nuovo comune per i socket);
- l'apertura di un file di dispositivo che richiede operazioni non immediate per una risposta (ad esempio l'apertura di un nastro che deve essere riavvolto);
- le operazioni eseguite con `ioctl` che non è detto possano essere eseguite immediatamente;
- l'uso di funzioni di intercomunicazione fra processi (vedi cap. 11) che si bloccano in attesa di risposte da altri processi;
- l'uso della funzione `pause` (vedi sez. 7.3.5) e le analoghe `sigsuspend`, `sigtimedwait`, e `sigwaitinfo` (vedi sez. 7.5.1), usate appunto per attendere l'arrivo di un segnale;
- l'uso delle funzioni associate al *file locking* (vedi sez. 10.1)
- l'uso della funzione `wait` e le analoghe funzioni di attesa se nessun processo figlio è ancora terminato.

In questo caso si pone il problema di cosa fare una volta che il gestore sia ritornato. La scelta originaria dei primi Unix era quella di far ritornare anche la *system call* restituendo l'errore di `EINTR`. Questa è a tutt'oggi una scelta corrente, ma comporta che i programmi che usano dei gestori controllino lo stato di uscita delle funzioni che eseguono una *system call* lenta per ripeterne la chiamata qualora l'errore fosse questo.

Dimenticarsi di richiamare una *system call* interrotta da un segnale è un errore comune, tanto che la *glibc* provvede una macro `TEMP_FAILURE_RETRY(expr)` che esegue l'operazione automaticamente, ripetendo l'esecuzione dell'espressione `expr` fintanto che il risultato non è diverso dall'uscita con un errore `EINTR`.

La soluzione è comunque poco elegante e BSD ha scelto un approccio molto diverso, che è quello di fare ripartire automaticamente una *system call* interrotta invece di farla fallire. In questo caso ovviamente non c'è bisogno di preoccuparsi di controllare il codice di errore; si perde però la possibilità di eseguire azioni specifiche all'occorrenza di questa particolare condizione.

Linux e la *glibc* consentono di utilizzare entrambi gli approcci, attraverso una opportuna opzione di `sigaction` (vedi sez. 7.4.3). È da chiarire comunque che nel caso di interruzione nel mezzo di un trasferimento parziale di dati, le *system call* ritornano sempre indicando i byte trasferiti.

Si tenga presente però che alcune *system call* vengono comunque interrotte con un errore di `EINTR` indipendentemente dal fatto che ne possa essere stato richiesto il riavvio automatico, queste funzioni sono:

- le funzioni di attesa di un segnale: `pause` (vedi sez. 7.3.5) o `sigsuspend`, `sigtimedwait`, e `sigwaitinfo` (vedi sez. 7.5.1).
- le funzioni di attesa dell'*I/O multiplexing* (vedi sez. 10.2) come `select`, `pselect`, `poll`, `ppoll`, `epoll_wait` e `epoll_pwait`.
- le funzioni del System V IPC che prevedono attese: `msgrcv`, `msgsnd` (vedi sez. 11.2.4), `semop` e `semtimedop` (vedi sez. 11.2.5).
- le funzioni per la messa in attesa di un processo come `usleep`, `nanosleep` (vedi sez. 7.3.5) e `clock_nanosleep` (vedi sez. 7.5.2).

- le funzioni che operano sui socket quando è stato impostato un *timeout* sugli stessi con `setsockopt` (vedi sez. 16.2.2) ed in particolare `accept`, `recv`, `recvfrom`, `recvmsg` per un *timeout* in ricezione e `connect`, `send`, `sendto` e `sendmsg` per un *timeout* in trasmissione.

7.3.2 L'installazione di un gestore

L'interfaccia più semplice per la gestione dei segnali è costituita dalla funzione di sistema `signal` che è definita fin dallo standard ANSI C. Quest'ultimo però non considera sistemi multitasking, per cui la definizione è tanto vaga da essere del tutto inutile in un sistema Unix. Per questo motivo ogni implementazione successiva ne ha modificato e ridefinito il comportamento, pur mantenendone immutato il prototipo¹ che è:

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler)
                                Installa un gestore di segnale (signal handler).
```

La funzione ritorna il precedente gestore in caso di successo in caso di successo e `SIG_ERR` per un errore, nel qual caso `errno` assumerà il valore:

`EINVAL` il numero di segnale `signum` non è valido.

In questa definizione per l'argomento `handler` che indica il gestore da installare si è usato un tipo di dato, `sighandler_t`, che è una estensione GNU, definita dalla *glibc*, che permette di riscrivere il prototipo di `signal` nella forma appena vista, molto più leggibile di quanto non sia la versione originaria, che di norma è definita come:

```
void ( *signal(int signum, void (*handler)(int)) )(int)
```

questa infatti, per la complessità della sintassi del C quando si vanno a trattare puntatori a funzioni, è molto meno comprensibile. Da un confronto con il precedente prototipo si può dedurre la definizione di `sighandler_t` che è:

```
typedef void (* sighandler_t)(int)
```

e cioè un puntatore ad una funzione `void` (cioè senza valore di ritorno) che prende un argomento di tipo `int`. Si noti come si devono usare le parentesi intorno al nome della funzione per via delle precedenze degli operatori del C, senza di esse si sarebbe definita una funzione che ritorna un puntatore a `void` e non un puntatore ad una funzione `void`.

La funzione `signal` quindi restituisce e prende come secondo argomento un puntatore a una funzione di questo tipo, che è appunto la funzione che verrà usata come gestore del segnale. Il numero di segnale passato nell'argomento `signum` può essere indicato direttamente con una delle costanti definite in sez. 7.2.1.

L'argomento `handler` che indica il gestore invece, oltre all'indirizzo della funzione da chiamare all'occorrenza del segnale, può assumere anche i due valori costanti `SIG_IGN` e `SIG_DFL`. Il primo indica che il segnale deve essere ignorato. Il secondo ripristina l'azione predefinita, e serve a tornare al comportamento di default quando non si intende più gestire direttamente un segnale.

Si ricordi però che i due segnali `SIGKILL` e `SIGSTOP` non possono essere né ignorati né intercettati e per loro l'uso di `signal` non ha alcun effetto, qualunque cosa si specifichi nell'argomento `handler`.

¹in realtà in alcune vecchie implementazioni (SVr4 e 4.3+BSD in particolare) vengono usati alcuni argomenti aggiuntivi per definire il comportamento della funzione, vedremo in sez. 7.4.3 che questo è possibile usando la funzione `sigaction`.

La funzione restituisce l'indirizzo dell'azione precedente, che può essere salvato per poterlo ripristinare (con un'altra chiamata a `signal`) in un secondo tempo. Si ricordi che se si imposta come azione `SIG_IGN` o si imposta `SIG_DFL` per un segnale la cui azione predefinita è di essere ignorato, tutti i segnali pendenti saranno scartati, e non verranno mai notificati.

L'uso di `signal` è soggetto a problemi di compatibilità, dato che essa si comporta in maniera diversa per sistemi derivati da BSD o da System V. In questi ultimi infatti la funzione è conforme al comportamento originale dei primi Unix in cui il gestore viene disinstallato alla sua chiamata secondo la semantica inaffidabile; anche Linux seguiva questa convenzione con le vecchie librerie del C come la *libc4* e la *libc5*.²

Al contrario BSD segue la semantica affidabile, non disinstallando il gestore e bloccando il segnale durante l'esecuzione dello stesso. Con l'utilizzo della *glibc* dalla versione 2 anche Linux è passato a questo comportamento. Il comportamento della versione originale della funzione, il cui uso è deprecato per i motivi visti in sez. 7.1.2, può essere ottenuto chiamando `sysv_signal`, una volta che si sia definita la macro `_XOPEN_SOURCE`.

In generale, per evitare questi problemi e per le possibili differenze nella semantica fra versioni diverse di kernel, l'uso di `signal` è sempre da evitare, visto che tra l'altro la funzione ha un comportamento indefinito in caso di processi multi-*thread*; l'unico utilizzo sicuro della funzione è con `SIG_IGN` e `SIG_DFL`, in tutti gli altri casi si deve usare `sigaction`.

Infine si deve tenere presente che su Linux, seguendo lo standard POSIX, il comportamento di un processo che ignora i segnali `SIGFPE`, `SIGILL`, o `SIGSEGV`, qualora questi non originino da una chiamata ad una `kill` o altra funzione affine, è indefinito. Un gestore che ritorna da questi segnali può dare luogo ad un ciclo infinito.

7.3.3 Le funzioni per l'invio di segnali

Come accennato in sez. 7.1.3 un segnale può anche essere generato direttamente nell'esecuzione di un programma, attraverso la chiamata ad una opportuna *system call*. Le funzioni che si utilizzano di solito per inviare un segnale generico ad un processo sono `raise` e `kill`.

La funzione `raise`, definita dallo standard ANSI C, serve per inviare un segnale al processo corrente,³ il suo prototipo è:

```
#include <signal.h>
int raise(int sig)
```

Invia un segnale al processo corrente.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà il valore:

`EINVAL` il segnale `sig` non è valido.

Il valore di `sig` specifica il segnale che si vuole inviare e può essere specificato con una delle costanti illustrate in tab. 7.1. In genere questa funzione viene usata per riprodurre il comportamento predefinito di un segnale che sia stato intercettato. In questo caso, una volta eseguite le operazioni volute, il gestore dovrà prima reinstallare l'azione predefinita, per poi attivarla chiamando `raise`.

In realtà `raise` è una funzione di libreria, che per i processi ordinari veniva implementata (nelle versioni più recenti del kernel viene usata `tkill` che vedremo in sez. 12.4.3) attraverso

²nelle *libc5* esisteva però la possibilità di includere `bsd/signal.h` al posto di `signal.h`, nel qual caso la funzione `signal` era ridefinita per seguire la semantica affidabile usata da BSD.

³non prevedendo la presenza di un sistema multiutente lo standard ANSI C non poteva che definire una funzione che invia il segnale al programma in esecuzione, nel caso di Linux questa viene implementata come funzione di compatibilità.

la funzione di sistema `kill` che è quella che consente effettivamente di inviare un segnale generico ad un processo, il suo prototipo è:

<pre>#include <sys/types.h> #include <signal.h> int kill(pid_t pid, int sig)</pre>	
	Invia un segnale ad uno o più processi.
<p>La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EINVAL</code> il segnale specificato non esiste.</p> <p><code>EPERM</code> non si hanno privilegi sufficienti ad inviare il segnale.</p> <p><code>ESRCH</code> il processo o il gruppo di processi indicato non esiste.</p>	

La funzione invia il segnale specificato dall'argomento `sig` al processo o ai processi specificati con l'argomento `pid`. Lo standard POSIX prevede che il valore 0 per `sig` sia usato per specificare il segnale nullo. Se la funzione viene chiamata con questo valore non viene inviato nessun segnale, ma viene eseguito il controllo degli errori, in tal caso si otterrà un errore `EPERM` se non si hanno i permessi necessari ed un errore `ESRCH` se il processo o i processi specificati con `pid` non esistono.

Valore	Significato
> 0	Il segnale è mandato al processo con <i>PID</i> uguale a <code>pid</code> .
0	Il segnale è mandato ad ogni processo del <i>process group</i> (vedi sez. 8.1.2) del chiamante.
-1	Il segnale è mandato ad ogni processo (eccetto <code>init</code>).
< -1	Il segnale è mandato ad ogni processo del <i>process group</i> con <i>PGID</i> uguale a <code> pid </code> .

Tabella 7.4: Valori dell'argomento `pid` per la funzione `kill`.

A seconda del valore dell'argomento `pid` si può inviare il segnale ad uno specifico processo, ad un *process group* (vedi sez. 8.1.2) o a tutti i processi, secondo quanto illustrato in tab. 7.4 che riporta i valori possibili per questo argomento. Si tenga conto però che il sistema ricicla i *PID* (come accennato in sez. 3.1.2) per cui l'esistenza di un processo non significa che esso sia realmente quello a cui si intendeva mandare il segnale (torneremo su questo in sez. 7.5.4).

Indipendentemente dalla funzione specifica che viene usata solo l'amministratore può inviare un segnale ad un processo qualunque, in tutti gli altri casi l'*UID* reale o l'*UID* effettivo del processo chiamante devono corrispondere all'*UID* reale o all'*UID* salvato della destinazione.⁴ Fa eccezione il caso in cui il segnale inviato sia `SIGCONT`, nel quale occorre anche che entrambi i processi appartengano alla stessa sessione.

Si tenga presente che, per il ruolo fondamentale che riveste nel sistema, non è possibile inviare al processo 1 (cioè a `init`) segnali per i quali esso non abbia un gestore installato. Infine, seguendo le specifiche POSIX 1003.1-2001, l'uso della chiamata `kill(-1, sig)` comporta che il segnale sia inviato (con la solita eccezione di `init`) a tutti i processi per i quali i permessi lo consentano. Lo standard permette comunque alle varie implementazioni di escludere alcuni processi specifici: nel caso in questione Linux non invia il segnale al processo che ha effettuato la chiamata.

Si noti pertanto che la funzione `raise(sig)` può essere definita in termini di `kill`, ed è sostanzialmente equivalente ad una `kill(getpid(), sig)`. Siccome `raise`, che è definita nello standard ISO C, non esiste in alcune vecchie versioni di Unix, in generale l'uso di `kill` finisce

⁴questo a partire dal kernel 1.3.78, seguendo lo standard POSIX.1; in precedenza il comportamento era diverso, gli interessati alla storia possono consultare la pagina di manuale della funzione.

per essere più portabile. Una seconda funzione che può essere definita in termini di `kill` è `killpg`, il suo prototipo è:

<pre>#include <signal.h> int killpg(pid_t pidgrp, int signal)</pre>	Invia un segnale ad un <i>process group</i> .
La funzione ritorna 0 in caso di successo e -1 per un errore, e gli errori sono gli stessi di <code>kill</code> .	

La funzione invia il segnale `signal` al *process group* il cui *PGID* (vedi sez. 8.1.2) è indicato dall'argomento `pidgrp`, che deve essere un intero positivo. Il suo utilizzo è sostanzialmente equivalente all'esecuzione di `kill(-pidgrp, signal)`.

Oltre alle precedenti funzioni di base, vedremo più avanti che esistono altre funzioni per inviare segnali generici, come `sigqueue` per i segnali *real-time* (vedi sez. 7.5.1) e le specifiche funzioni per i *thread* che tratteremo in sez. 12.4.3.

Esiste però un'ultima funzione che permette l'invio diretto di un segnale che vale la pena di trattare a parte per le sue peculiarità. La funzione in questione è `abort` che, come accennato in sez. 3.1.4, permette di abortire l'esecuzione di un programma tramite l'invio del segnale `SIGABRT`. Il suo prototipo è:

<pre>#include <stdlib.h> void abort(void)</pre>	Abortisce il processo corrente.
La funzione non ritorna, il processo viene terminato.	

La differenza fra questa funzione e l'uso di `raise` o di un'altra funzione per l'invio di `SIGABRT` è che anche se il segnale è bloccato o ignorato, la funzione ha effetto lo stesso. Il segnale può però essere intercettato per effettuare eventuali operazioni di chiusura prima della terminazione del processo.

Lo standard ANSI C richiede inoltre che anche se il gestore ritorna, la funzione non ritorni comunque. Lo standard POSIX.1 va oltre e richiede che se il processo non viene terminato direttamente dal gestore sia la stessa `abort` a farlo al ritorno dello stesso. Inoltre, sempre seguendo lo standard POSIX, prima della terminazione tutti i file aperti e gli stream saranno chiusi ed i buffer scaricati su disco. Non verranno invece eseguite le eventuali funzioni registrate con `atexit` e `on_exit`.

7.3.4 Le funzioni di allarme ed i *timer*

Un caso particolare di segnali generati a richiesta è quello che riguarda i vari segnali usati per la temporizzazione, per ciascuno di essi infatti sono previste delle funzioni specifiche che ne effettuino l'invio. La più comune, e la più semplice, delle funzioni usate per la temporizzazione è la funzione di sistema `alarm`, il cui prototipo è:

<pre>#include <unistd.h> unsigned int alarm(unsigned int seconds)</pre>	Predisporre l'invio di un allarme.
La funzione ritorna il numero di secondi rimanenti ad un precedente allarme, o 0 se non c'erano allarmi pendenti, non sono previste condizioni di errore.	

La funzione fornisce un meccanismo che consente ad un processo di predisporre un'interruzione nel futuro, ad esempio per effettuare una qualche operazione dopo un certo periodo di tempo, programmando l'emissione di un segnale (nel caso in questione `SIGALRM`) dopo il numero di secondi specificato dall'argomento `seconds`. Se si specifica per `seconds` un valore

nullo non verrà inviato nessun segnale. Siccome alla chiamata viene cancellato ogni precedente allarme, questo valore può essere usato per cancellare una programmazione precedente.

La funzione inoltre ritorna il numero di secondi rimanenti all'invio dell'allarme programmato in precedenza. In questo modo è possibile controllare se non si è cancellato un precedente allarme e predisporre eventuali misure che permettano di gestire il caso in cui servono più interruzioni.

In sez. 6.4.1 abbiamo visto che ad ogni processo sono associati tre tempi diversi: il *clock time*, l'*user time* ed il *system time*. Per poterli calcolare il kernel mantiene per ciascun processo tre diversi timer:

- un *real-time timer* che calcola il tempo reale trascorso (che corrisponde al *clock time*). La scadenza di questo timer provoca l'emissione di SIGALRM;
- un *virtual timer* che calcola il tempo di processore usato dal processo in *user space* (che corrisponde all'*user time*). La scadenza di questo timer provoca l'emissione di SIGVTALRM;
- un *profiling timer* che calcola la somma dei tempi di processore utilizzati direttamente dal processo in *user space*, e dal kernel nelle *system call* ad esso relative (che corrisponde a quello che in sez. 6.4.1 abbiamo chiamato *processor time*). La scadenza di questo timer provoca l'emissione di SIGPROF.

Il timer usato da `alarm` è il *clock time*, e corrisponde cioè al tempo reale. La funzione come abbiamo visto è molto semplice, ma proprio per questo presenta numerosi limiti: non consente di usare gli altri timer, non può specificare intervalli di tempo con precisione maggiore del secondo e genera il segnale una sola volta.

Per ovviare a questi limiti Linux deriva da BSD la funzione `setitimer` che permette di usare un timer qualunque e l'invio di segnali periodici, al costo però di una maggiore complessità d'uso e di una minore portabilità. Il suo prototipo è:

```
#include <sys/time.h>
int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue)
                                                    Predispose l'invio di un segnale di allarme.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori EINVAL o EFAULT nel loro significato generico.

La funzione predispose l'invio di un segnale di allarme alla scadenza dell'intervallo indicato dall'argomento `value`. Il valore dell'argomento `which` permette di specificare quale dei tre timer illustrati in precedenza usare; i possibili valori sono riportati in tab. 7.5.

Valore	Timer
ITIMER_REAL	<i>real-time timer</i>
ITIMER_VIRTUAL	<i>virtual timer</i>
ITIMER_PROF	<i>profiling timer</i>

Tabella 7.5: Valori dell'argomento `which` per la funzione `setitimer`.

Il valore della struttura specificata `value` viene usato per impostare il timer, se il puntatore `ovalue` non è nullo il precedente valore viene salvato qui. I valori dei timer devono essere indicati attraverso una struttura `itimerval`, definita in fig. 4.13.

La struttura è composta da due membri, il primo, `it_interval` definisce il periodo del timer; il secondo, `it_value` il tempo mancante alla scadenza. Entrambi esprimono i tempi tramite una struttura `timeval` che permette una precisione fino al microsecondo.

Ciascun timer decrementa il valore di `it_value` fino a zero, poi invia il segnale e reimposta `it_value` al valore di `it_interval`, in questo modo il ciclo verrà ripetuto; se invece il valore di `it_interval` è nullo il timer si ferma.

```

struct itimerval
{
    struct timeval it_interval; /* next value */
    struct timeval it_value;   /* current value */
};

```

Figura 7.2: La struttura `itimerval`, che definisce i valori dei timer di sistema.

L'uso di `setitimer` consente dunque un controllo completo di tutte le caratteristiche dei timer, ed in effetti la stessa `alarm`, benché definita direttamente nello standard POSIX.1, può a sua volta essere espressa in termini di `setitimer`, come evidenziato dal manuale della *glibc* [?] che ne riporta la definizione mostrata in fig. 7.3.⁵

```

unsigned int alarm(unsigned int seconds)
{
    struct itimerval old, new;
    new.it_interval.tv_usec = 0;
    new.it_interval.tv_sec = 0;
    new.it_value.tv_usec = 0;
    new.it_value.tv_sec = (long int) seconds;
    if (setitimer(ITIMER_REAL, &new, &old) < 0) {
        return 0;
    }
    else {
        return old.it_value.tv_sec;
    }
}

```

Figura 7.3: Definizione di `alarm` in termini di `setitimer`.

Si deve comunque tenere presente che fino al kernel 2.6.16 la precisione di queste funzioni era limitata dalla frequenza del timer di sistema, determinato dal valore della costante `HZ` di cui abbiamo già parlato in sez. 3.1.1, in quanto le temporizzazioni erano calcolate in numero di interruzioni del timer (i cosiddetti “*jiffies*”), ed era assicurato soltanto che il segnale non sarebbe stato mai generato prima della scadenza programmata (l'arrotondamento cioè era effettuato per eccesso).⁶

L'uso del contatore dei *jiffies*, un intero a 32 bit nella maggior parte dei casi, comportava inoltre l'impossibilità di specificare tempi molto lunghi. superiori al valore della costante `MAX_SEC_IN_JIFFIES`, pari, nel caso di default di un valore di `HZ` di 250, a circa 99 giorni e mezzo. Con il cambiamento della rappresentazione effettuato nel kernel 2.6.16 questo problema è scomparso e con l'introduzione dei timer ad alta risoluzione (vedi sez. 7.5.2) nel kernel 2.6.21 la precisione è diventata quella fornita dall'hardware disponibile.

Una seconda causa di potenziali ritardi è che il segnale viene generato alla scadenza del timer, ma poi deve essere consegnato al processo; se quest'ultimo è attivo (questo è sempre vero per `ITIMER_VIRTUAL`) la consegna è immediata, altrimenti può esserci un ulteriore ritardo che può variare a seconda del carico del sistema.

⁵questo comporta anche che non è il caso di mescolare chiamate ad `abort` e a `setitimer`.

⁶questo in realtà non è del tutto vero a causa di un bug, presente fino al kernel 2.6.12, che in certe circostanze causava l'emissione del segnale con un arrotondamento per difetto.

Questo ha una conseguenza che può indurre ad errori molto subdoli, si tenga conto poi che in caso di sistema molto carico, si può avere il caso patologico in cui un timer scade prima che il segnale di una precedente scadenza sia stato consegnato. In questo caso, per il comportamento dei segnali descritto in sez. 7.3.6, un solo segnale sarà consegnato. Per questo oggi l'uso di questa funzione è deprecato a favore degli *high-resolution timer* e della cosiddetta *POSIX Timer API*, che tratteremo in sez. 7.5.2.

Dato che sia `alarm` che `setitimer` non consentono di leggere il valore corrente di un timer senza modificarlo, è possibile usare la funzione `getitimer`, il cui prototipo è:

```
#include <sys/time.h>
int getitimer(int which, struct itimerval *value)
```

Legge il valore di un timer.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà gli stessi valori di `getitimer`.

La funzione legge nella struttura `itimerval` puntata da `value` il valore del timer specificato da `which` ed i suoi argomenti hanno lo stesso significato e formato di quelli di `setitimer`.

7.3.5 Le funzioni di pausa e attesa

Sono parecchie le occasioni in cui si può avere necessità di sospendere temporaneamente l'esecuzione di un processo. Nei sistemi più elementari in genere questo veniva fatto con un ciclo di attesa in cui il programma ripete una operazione un numero sufficiente di volte per far passare il tempo richiesto.

Ma in un sistema multitasking un ciclo di attesa è solo un inutile spreco di tempo di processore dato che altri programmi possono essere eseguiti nel frattempo, per questo ci sono delle apposite funzioni che permettono di mantenere un processo in attesa per il tempo voluto, senza impegnare il processore. In pratica si tratta di funzioni che permettono di portare esplicitamente il processo nello stato di *sleep* (si ricordi quanto illustrato in tab. 3.8) per un certo periodo di tempo.

La prima di queste è la funzione di sistema `pause`, che viene usata per mettere un processo in attesa per un periodo di tempo indefinito, fino all'arrivo di un segnale, il suo prototipo è:

```
#include <unistd.h>
int pause(void)
```

Pone il processo in pausa fino al ricevimento di un segnale.

La funzione ritorna solo dopo che un segnale è stato ricevuto ed il relativo gestore è ritornato, nel qual caso restituisce -1 e `errno` assume il valore `EINTR`.

La funzione ritorna sempre con una condizione di errore, dato che il successo sarebbe quello di continuare ad aspettare indefinitamente. In genere si usa questa funzione quando si vuole mettere un processo in attesa di un qualche evento specifico che non è sotto il suo diretto controllo, ad esempio la si può usare per interrompere l'esecuzione del processo fino all'arrivo di un segnale inviato da un altro processo.

Quando invece si vuole fare attendere un processo per un intervallo di tempo già noto in partenza, lo standard POSIX.1 prevede una funzione di attesa specifica, `sleep`, il cui prototipo è:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds)
```

Pone il processo in pausa per un tempo in secondi.

La funzione ritorna 0 se l'attesa viene completata o il numero di secondi restanti se viene interrotta da un segnale, non sono previsti codici di errore.

La funzione pone il processo in stato di *sleep* per il numero di secondi specificato dall'argomento *seconds*, a meno di non essere interrotta da un segnale. Alla terminazione del periodo di tempo indicato la funzione ritorna riportando il processo in stato *runnable* così che questo possa riprendere l'esecuzione.

In caso di interruzione della funzione non è una buona idea ripetere la chiamata per il tempo rimanente restituito dalla stessa, in quanto la riattivazione del processo può avvenire in un qualunque momento, ma il valore restituito sarà sempre arrotondato al secondo. Questo può avere la conseguenza che se la successione dei segnali è particolarmente sfortunata e le differenze si accumulano, si possono avere ritardi anche di parecchi secondi rispetto a quanto programmato inizialmente. In genere la scelta più sicura in questo caso è quella di stabilire un termine per l'attesa, e ricalcolare tutte le volte il numero di secondi che restano da aspettare.

Si tenga presente che alcune implementazioni l'uso di *sleep* può avere conflitti con quello di *SIGALRM*, dato che la funzione può essere realizzata con l'uso di *pause* e *alarm*, in una maniera analoga a quella dell'esempio che vedremo in sez. 7.4.1. In tal caso mescolare chiamate di *alarm* e *sleep* o modificare l'azione associata *SIGALRM*, può portare a dei risultati indefiniti. Nel caso della *glibc* è stata usata una implementazione completamente indipendente e questi problemi non ci sono, ma un programma portabile non può fare questa assunzione.

La granularità di *sleep* permette di specificare attese soltanto in secondi, per questo sia sotto BSD4.3 che in SUSv2 è stata definita un'altra funzione con una precisione teorica del microsecondo. I due standard hanno delle definizioni diverse, ma la *glibc* segue (secondo la pagina di manuale almeno dalla versione 2.2.2) quella di SUSv2 per cui la funzione *usleep* (dove la *u* è intesa come sostituzione di μ), ha il seguente prototipo:

```
#include <unistd.h>
int usleep(unsigned long usec)
```

Pone il processo in pausa per un tempo in microsecondi.

La funzione ritorna 0 se l'attesa viene completata e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EINTR la funzione è stata interrotta da un segnale.

EINVAL si è indicato un valore di *usec* maggiore di 1000000.

Anche questa funzione, a seconda delle implementazioni, può presentare problemi nell'interazione con *alarm* e *SIGALRM*, per questo motivo, pur essendovi citata, nello standard POSIX.1-2001 viene deprecata in favore della nuova funzione di sistema *nanosleep*, il cui prototipo è:

```
#include <unistd.h>
int nanosleep(const struct timespec *req, struct timespec *rem)
```

Pone il processo in pausa per un intervallo di tempo.

La funzione ritorna 0 se l'attesa viene completata e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EINTR la funzione è stata interrotta da un segnale.

EINVAL si è specificato un numero di secondi negativo o un numero di nanosecondi maggiore di 999.999.999.

La funzione pone il processo in pausa portandolo nello stato di *sleep* per il tempo specificato dall'argomento *req*, ed in caso di interruzione restituisce il tempo restante nell'argomento *rem*. Lo standard richiede che la funzione sia implementata in maniera del tutto indipendente da *alarm*, e nel caso di Linux questo è fatto utilizzando direttamente il timer del kernel. Lo standard richiede inoltre che la funzione sia utilizzabile senza interferenze con l'uso di *SIGALRM*. La funzione prende come argomenti delle strutture di tipo *timespec*, la cui definizione è

riportata in fig. 4.16, il che permette di specificare un tempo con una precisione teorica fino al nanosecondo.

La funzione risolve anche il problema di proseguire l'attesa dopo l'interruzione dovuta ad un segnale; infatti in tal caso in `rem` viene restituito il tempo rimanente rispetto a quanto richiesto inizialmente,⁷ e basta richiamare la funzione per completare l'attesa.

Anche qui però occorre tenere presente che i tempi sono arrotondati, per cui la precisione, per quanto migliore di quella ottenibile con `sleep`, è relativa e in caso di molte interruzioni si può avere una deriva, per questo esiste la funzione `clock_nanosleep` (vedi sez. 7.5.2) che permette di specificare un tempo assoluto anziché un tempo relativo.

Chiaramente, anche se il tempo può essere specificato con risoluzioni fino al nanosecondo, la precisione di `nanosleep` è determinata dalla risoluzione temporale del timer di sistema. Perciò la funzione attenderà comunque il tempo specificato, ma prima che il processo possa tornare ad essere eseguito occorrerà almeno attendere la successiva interruzione del timer di sistema, cioè un tempo che a seconda dei casi può arrivare fino a 1/HZ, (sempre che il sistema sia scarico ed il processa venga immediatamente rimesso in esecuzione). Per questo motivo il valore restituito in `rem` è sempre arrotondato al multiplo successivo di 1/HZ.

Con i kernel della serie 2.4 in realtà era possibile ottenere anche pause più precise del centesimo di secondo usando politiche di *scheduling real-time* come `SCHED_FIFO` o `SCHED_RR` (vedi sez. 3.3.3); in tal caso infatti il calcolo sul numero di interruzioni del timer veniva evitato utilizzando direttamente un ciclo di attesa con cui si raggiungevano pause fino ai 2 ms con precisioni del μ s. Questa estensione è stata rimossa con i kernel della serie 2.6, che consentono una risoluzione più alta del timer di sistema; inoltre a partire dal kernel 2.6.21, `nanosleep` può avvalersi del supporto dei timer ad alta risoluzione, ottenendo la massima precisione disponibile sull'hardware della propria macchina.

7.3.6 Un esempio elementare

Un semplice esempio per illustrare il funzionamento di un gestore di segnale è quello della gestione di `SIGCHLD`. Abbiamo visto in sez. 3.1.4 che una delle azioni eseguite dal kernel alla conclusione di un processo è quella di inviare questo segnale al padre. In generale dunque, quando non interessa elaborare lo stato di uscita di un processo, si può completare la gestione della terminazione installando un gestore per `SIGCHLD` il cui unico compito sia quello di chiamare `waitpid` per completare la procedura di terminazione in modo da evitare la formazione di *zombie*.⁸

In fig. 7.4 è mostrato il codice contenente una implementazione generica di una funzione di gestione per `SIGCHLD`, (che si trova nei sorgenti allegati nel file `SigHand.c`); se ripetiamo i test di sez. 3.1.4, invocando `forktest` con l'opzione `-s` (che si limita ad effettuare l'installazione di questa funzione come gestore di `SIGCHLD`) potremo verificare che non si ha più la creazione di *zombie*.

Il codice del gestore è di lettura immediata, come buona norma di programmazione (si ricordi quanto accennato sez. 6.5.1) si comincia (6-7) con il salvare lo stato corrente di `errno`, in modo da poterlo ripristinare prima del ritorno del gestore (16-17). In questo modo si preserva il valore della variabile visto dal corso di esecuzione principale del processo, che altrimenti sarebbe sovrascritto dal valore restituito nella successiva chiamata di `waitpid`.

⁷con l'eccezione, valida solo nei kernel della serie 2.4, in cui, per i processi riavviati dopo essere stati fermati da un segnale, il tempo passato in stato `T` non viene considerato nel calcolo della rimanenza.

⁸si ricordi comunque che dal kernel 2.6 seguendo lo standard POSIX.1-2001 per evitare di dover ricevere gli stati di uscita che non interessano basta impostare come azione predefinita quella di ignorare `SIGCHLD`, nel qual caso viene assunta la semantica di System V, in cui il segnale non viene inviato, il sistema non genera *zombie* e lo stato di terminazione viene scartato senza dover chiamare una `wait`.

```
1 void HandSigCHLD(int sig)
2 {
3     int errno_save;
4     int status;
5     pid_t pid;
6     /* save errno current value */
7     errno_save = errno;
8     /* loop until no */
9     do {
10         errno = 0;
11         pid = waitpid(WAIT_ANY, &status, WNOHANG);
12     } while (pid > 0);
13     /* restore errno value */
14     errno = errno_save;
15     /* return */
16     return;
17 }
```

Figura 7.4: Codice di una funzione generica di gestione per il segnale SIGCHLD.

Il compito principale del gestore è quello di ricevere lo stato di terminazione del processo, cosa che viene eseguita nel ciclo in (9-15). Il ciclo è necessario a causa di una caratteristica fondamentale della gestione dei segnali: abbiamo già accennato come fra la generazione di un segnale e l'esecuzione del gestore possa passare un certo lasso di tempo e niente ci assicura che il gestore venga eseguito prima della generazione di ulteriori segnali dello stesso tipo. In questo caso normalmente i segnali successivi vengono “fusi” col primo ed al processo ne viene recapitato soltanto uno.

Questo può essere un caso comune proprio con SIGCHLD, qualora capitasse che molti processi figli terminino in rapida successione. Esso inoltre si presenta tutte le volte che un segnale viene bloccato: per quanti siano i segnali emessi durante il periodo di blocco, una volta che quest'ultimo sarà rimosso verrà recapitato un solo segnale.

Allora, nel caso della terminazione dei processi figli, se si chiamasse `waitpid` una sola volta, essa leggerebbe lo stato di terminazione per un solo processo, anche se i processi terminati sono più di uno, e gli altri resterebbero in stato di *zombie* per un tempo indefinito.

Per questo occorre ripetere la chiamata di `waitpid` fino a che essa non ritorni un valore nullo, segno che non resta nessun processo di cui si debba ancora ricevere lo stato di terminazione (si veda sez. 3.1.5 per la sintassi della funzione). Si noti anche come la funzione venga invocata con il parametro `WNOHANG` che permette di evitare il suo blocco quando tutti gli stati di terminazione sono stati ricevuti.

7.4 La gestione avanzata dei segnali

Le funzioni esaminate finora fanno riferimento alle modalità più elementari della gestione dei segnali; non si sono pertanto ancora prese in considerazione le tematiche più complesse, collegate alle varie *race condition* che i segnali possono generare e alla natura asincrona degli stessi.

Affronteremo queste problematiche in questa sezione, partendo da un esempio che le evidenzia, per poi prendere in esame le varie funzioni che permettono di risolvere i problemi più

complessi connessi alla programmazione con i segnali, fino a trattare le caratteristiche generali della gestione dei medesimi nella casistica ordinaria.

7.4.1 Alcune problematiche aperte

Come accennato in sez. 7.3.5 è possibile implementare `sleep` a partire dall'uso di pause e `alarm`. A prima vista questo può sembrare di implementazione immediata; ad esempio una semplice versione di `sleep` potrebbe essere quella illustrata in fig. 7.5.

```

1 void alarm_hand(int sig) {
2     /* check if the signal is the right one */
3     if (sig != SIGALRM) { /* if not exit with error */
4         printf("Something wrong, handler for SIGALRM\n");
5         exit(1);
6     } else { /* do nothing, just interrupt pause */
7         return;
8     }
9 }
10 unsigned int sleep(unsigned int seconds)
11 {
12     sighandler_t prev_handler;
13     /* install and check new handler */
14     if ((prev_handler = signal(SIGALRM, alarm_hand)) == SIG_ERR) {
15         printf("Cannot set handler for alarm\n");
16         exit(-1);
17     }
18     /* set alarm and go to sleep */
19     alarm(seconds);
20     pause();
21     /* restore previous signal handler */
22     signal(SIGALRM, prev_handler);
23     /* return remaining time */
24     return alarm(0);
25 }

```

Figura 7.5: Una implementazione pericolosa di `sleep`.

Dato che è nostra intenzione utilizzare `SIGALRM` il primo passo della nostra implementazione sarà quello di installare il relativo gestore salvando il precedente (14-17). Si effettuerà poi una chiamata ad `alarm` per specificare il tempo d'attesa per l'invio del segnale a cui segue la chiamata a `pause` per fermare il programma (18-20) fino alla sua ricezione. Al ritorno di `pause`, causato dal ritorno del gestore (1-9), si ripristina il gestore originario (21-22) restituendo l'eventuale tempo rimanente (23-24) che potrà essere diverso da zero qualora l'interruzione di `pause` venisse causata da un altro segnale.

Questo codice però, a parte il non gestire il caso in cui si è avuta una precedente chiamata a `alarm` (che si è tralasciato per brevità), presenta una pericolosa *race condition*. Infatti, se il processo viene interrotto fra la chiamata di `alarm` e `pause`, può capitare (ad esempio se il sistema è molto carico) che il tempo di attesa scada prima dell'esecuzione di quest'ultima, cosicché essa sarebbe eseguita dopo l'arrivo di `SIGALRM`. In questo caso ci si troverebbe di fronte ad un *deadlock*, in quanto `pause` non verrebbe mai più interrotta (se non in caso di un altro segnale).

Questo problema può essere risolto (ed è la modalità con cui veniva fatto in SVr2) usando la funzione `longjmp` (vedi sez. 2.4.3) per uscire dal gestore. In questo modo, con una condizione

sullo stato di uscita di quest'ultima, si può evitare la chiamata a `pause`, usando un codice del tipo di quello riportato in fig. 7.6.

```

1 static jmp_buff alarm_return;
2 unsigned int sleep(unsigned int seconds)
3 {
4     sigandler_t prev_handler;
5     if ((prev_handler = signal(SIGALRM, alarm_hand)) == SIG_ERR) {
6         printf("Cannot set handler for alarm\n");
7         exit(1);
8     }
9     if (setjmp(alarm_return) == 0) { /* if not returning from handler */
10         alarm(second); /* call alarm */
11         pause(); /* then wait */
12     }
13     /* restore previous signal handler */
14     signal(SIGALRM, prev_handler);
15     /* remove alarm, return remaining time */
16     return alarm(0);
17 }
18 void alarm_hand(int sig)
19 {
20     /* check if the signal is the right one */
21     if (sig != SIGALRM) { /* if not exit with error */
22         printf("Something wrong, handler for SIGALRM\n");
23         exit(1);
24     } else { /* return in main after the call to pause */
25         longjmp(alarm_return, 1);
26     }
27 }

```

Figura 7.6: Una implementazione ancora malfunzionante di `sleep`.

In questo caso il gestore (18-27) non ritorna come in fig. 7.5, ma usa la funzione `longjmp` (25) per rientrare direttamente nel corpo principale del programma. Dato che in questo caso il valore di uscita che verrà restituito da `setjmp` è 1, grazie alla condizione impostata in (9-12) si potrà evitare comunque che `pause` sia chiamata a vuoto.

Ma anche questa implementazione comporta dei problemi, in questo caso infatti non viene gestita correttamente l'interazione con gli altri segnali. Se infatti il segnale di allarme interrompe un altro gestore, l'esecuzione non riprenderà nel gestore in questione, ma nel ciclo principale, interrompendone inopportuno l'esecuzione. Lo stesso tipo di problemi si presenterebbero se si volesse usare questa implementazione di `alarm` per stabilire un timeout su una qualunque *system call* bloccante.

Un secondo esempio dei problemi a cui si può andare incontro è quello in cui si usa un segnale per notificare una qualche forma di evento. In genere quello che si fa in questo caso è impostare all'interno del gestore un opportuno flag da controllare nel corpo principale del programma, con un codice del tipo di quello riportato in fig. 7.7.

La logica del programma è quella di impostare nel gestore una variabile globale preventivamente inizializzata nel programma principale ad un valore diverso (14-19). In questo modo dal corpo principale del programma si potrà determinare, osservando il contenuto di detta variabile, l'occorrenza o meno del segnale, ed eseguire le conseguenti azioni relative (6-11).

Questo è il tipico esempio di caso, già citato in sez. 3.4.2, in cui si genera una *race condition*. Infatti, in una situazione in cui un segnale è già arrivato (e quindi `flag` è già stata impostata

```
1 sig_atomic_t flag;
2 int main()
3 {
4     flag = 0;
5     ...
6     if (flag) {           /* test if signal occurred */
7         flag = 0;         /* reset flag */
8         do_response();    /* do things */
9     } else {
10        do_other();        /* do other things */
11    }
12    ...
13 }
14 void alarm_hand(int sig)
15 {
16     /* set the flag */
17     flag = 1;
18     return;
19 }
```

Figura 7.7: Un esempio non funzionante del codice per il controllo di un evento generato da un segnale.

ad 1 nel gestore) se un altro segnale arriva immediatamente dopo l'esecuzione del controllo (6) ma prima della cancellazione di `flag` fatta subito dopo (7), la sua occorrenza sarà perduta.

Questi esempi ci mostrano come per poter eseguire una gestione effettiva dei segnali occorrono delle funzioni più sofisticate di quelle finora illustrate. La funzione `signal` infatti ha la sua origine nell'interfaccia alquanto primitiva che venne adottata nei primi sistemi Unix, ma con questa funzione è sostanzialmente impossibile gestire in maniera adeguata di tutti i possibili aspetti con cui un processo deve reagire alla ricezione di un segnale.

7.4.2 Gli insiemi di segnali o *signal set*

Come evidenziato nel paragrafo precedente, le funzioni di gestione dei segnali originarie, nate con la semantica inaffidabile, hanno dei limiti non superabili; in particolare non è prevista nessuna funzione che permetta di gestire il blocco dei segnali o di verificare lo stato dei segnali pendenti.

Per questo motivo lo standard POSIX.1, insieme alla nuova semantica dei segnali ha introdotto una interfaccia di gestione completamente nuova, che permette di ottenere un controllo molto più dettagliato. In particolare lo standard ha introdotto un nuovo tipo di dato `sigset_t`, che permette di rappresentare un *insieme di segnali* (un *signal set*, come viene usualmente chiamato), tale tipo di dato viene usato per gestire il blocco dei segnali.

Inizialmente un *insieme di segnali* veniva rappresentato da un intero di dimensione opportuna, di solito pari al numero di bit dell'architettura della macchina, ciascun bit del quale era associato ad uno specifico segnale. Nel caso di architetture a 32 bit questo comporta un massimo di 32 segnali distinti e dato che a lungo questi sono stati sufficienti non c'era necessità di nessuna struttura più complicata, in questo modo era possibile implementare le operazioni direttamente con istruzioni elementari del processore.

Oggi questo non è più vero, in particolare con l'introduzione dei segnali *real-time* (che vedremo in sez. 7.5.1). Dato che in generale non si può fare conto sulle caratteristiche di una implementazione, perché non è detto che si disponga di un numero di bit sufficienti per mettere tutti i segnali in un intero, o perché in `sigset_t` possono essere immagazzinate ulteriori

informazioni, tutte le operazioni devono essere effettuate tramite le opportune funzioni di libreria che si curano di mascherare i dettagli di basso livello.

Lo standard POSIX.1 definisce cinque funzioni per la manipolazione degli insiemi di segnali. Le prime quattro, che consentono di manipolare i contenuti di un *signal set*, sono `sigemptyset`, `sigfillset`, `sigaddset` e `sigdelset`; i rispettivi prototipi sono:

```
#include <signal.h>
int sigemptyset(sigset_t *set)
                                Inizializza un insieme di segnali vuoto.

int sigfillset(sigset_t *set)
                                Inizializza un insieme di segnali pieno.

int sigaddset(sigset_t *set, int signum)
                                Aggiunge un segnale ad un insieme di segnali.

int sigdelset(sigset_t *set, int signum)
                                Rimuove un segnale da un insieme di segnali.
```

Le funzioni ritornano 0 in caso di successo, e -1 per un errore, nel qual caso `errno` assumerà il valore:

`EINVAL` `signum` non è un segnale valido.

Le prime due funzioni inizializzano l'insieme di segnali indicato dall'argomento `set` rispettivamente ad un contenuto vuoto (in cui cioè non c'è nessun segnale) e pieno (in cui cioè ci sono tutti i segnali). Le altre due funzioni consentono di inserire o rimuovere uno specifico segnale indicato con l'argomento `signum` in un insieme.

A queste funzioni si aggiunge l'ulteriore `sigismember`, che consente di verificare la presenza di un segnale in un insieme, il suo prototipo è:

```
#include <signal.h>
int sigismember(const sigset_t *set, int signum)
                                Controlla se un segnale è in un insieme di segnali.
```

La funzione ritorna 1 il segnale è nell'insieme e 0 altrimenti, e -1 per un errore, nel qual caso `errno` assumerà il valore `EINVAL` se si è specificato un puntatore `NULL`.

La *glibc* prevede inoltre altre funzioni non standardizzate, accessibili definendo la macro `_GNU_SOURCE`. La prima di queste è `sigisemptyset`, che consente di verificare un insieme è vuoto, il suo prototipo è:

```
#include <signal.h>
int sigisemptyset(sigset_t *set)
                                Controlla se un insieme di segnali è vuoto.
```

La funzione ritorna 1 l'insieme è vuoto e 0 altrimenti, non sono previste condizioni di errore.

Alla precedente si aggiungono altre due funzioni consentono di effettuare delle operazioni logiche con gli insiemi di segnali, esse sono `sigorset` e `sigandset`, ed i rispettivi prototipi sono:

```
#include <signal.h>
sigorset(sigset_t *dest, sigset_t *left, sigset_t *right)
                                Crea l'unione di due insieme di segnali.

sigandset(sigset_t *dest, sigset_t *left, sigset_t *right)
                                Crea l'intersezione di due insieme di segnali.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà il valore `EINVAL`.

In genere si usa un insieme di segnali per specificare quali segnali si vuole bloccare, o per riottenere dalle varie funzioni di gestione la maschera dei segnali attivi (vedi sez. 7.4.4). La modalità più comune, che è anche quella più portabile, prevede che possano essere definiti

aggiungendo i segnali voluti ad un insieme vuoto ottenuto con `sigemptyset` o togliendo quelli che non servono da un insieme completo ottenuto con `sigfillset`.

7.4.3 La funzione `sigaction`

Abbiamo già accennato in sez. 7.3.2 i problemi di compatibilità relativi all'uso di `signal`. Per ovviare a tutto questo lo standard POSIX.1 ha ridefinito completamente l'interfaccia per la gestione dei segnali, rendendola molto più flessibile e robusta, anche se leggermente più complessa.

La principale funzione di sistema prevista dall'interfaccia POSIX.1 per la gestione dei segnali è `sigaction`. Essa ha sostanzialmente lo stesso uso di `signal`, permette cioè di specificare le modalità con cui un segnale può essere gestito da un processo. Il suo prototipo è:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
                                                    Installa una nuova azione per un segnale.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT si sono specificati indirizzi non validi.

EINVAL si è specificato un numero di segnale invalido o si è cercato di installare il gestore per SIGKILL o SIGSTOP.

La funzione serve ad installare una nuova *azione* per il segnale indicato dall'argomento `signum`. Si parla di *azione* e non di *gestore* come nel caso di `signal`, in quanto la funzione consente di specificare le varie caratteristiche della risposta al segnale, non solo la funzione che verrà eseguita alla sua occorrenza.

Per questo motivo lo standard POSIX.1 raccomanda di usare sempre questa funzione al posto della precedente `signal`, che in genere viene ridefinita in termini di `sigaction`, in quanto la nuova interfaccia permette un controllo completo su tutti gli aspetti della gestione di un segnale, sia pure al prezzo di una maggiore complessità d'uso.

Se il puntatore `act` non è nullo, la funzione installa la nuova azione da esso specificata, se `oldact` non è nullo il valore dell'azione corrente viene restituito indietro. Questo permette (specificando `act` nullo e `oldact` non nullo) di superare uno dei limiti di `signal`, che non consente di ottenere l'azione corrente senza installarne una nuova. Se sia `act` che `oldact` sono nulli la funzione può essere utilizzata per verificare che il segnale indicato sia valido per la piattaforma che si sta usando (se non lo è darà errore).

```
struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

Figura 7.8: La struttura `sigaction`.

Entrambi i puntatori fanno riferimento alla struttura `sigaction`, tramite la quale si specificano tutte le caratteristiche dell'azione associata ad un segnale. Anch'essa è descritta dal-

lo standard POSIX.1 ed in Linux è definita secondo quanto riportato in fig. 7.8. Il campo `sa_restorer`, non previsto dallo standard, è obsoleto e non deve essere più usato.

Il campo `sa_mask` serve ad indicare l'insieme dei segnali che devono essere bloccati durante l'esecuzione del gestore, ad essi viene comunque sempre aggiunto il segnale che ne ha causato la chiamata, a meno che non si sia specificato con `sa_flag` un comportamento diverso. Quando il gestore ritorna la maschera dei segnali bloccati (vedi sez. 7.4.4) viene comunque ripristinata al valore precedente l'invocazione.

L'uso di questo campo permette ad esempio di risolvere il problema residuo dell'implementazione di `sleep` mostrata in fig. 7.6. In quel caso infatti se il segnale di allarme avesse interrotto un altro gestore questo non sarebbe stato eseguito correttamente, la cosa poteva essere prevenuta installando gli altri gestori usando `sa_mask` per bloccare `SIGALRM` durante la loro esecuzione. Il valore di `sa_flag` permette di specificare vari aspetti del comportamento di `sigaction`, e della reazione del processo ai vari segnali; i valori possibili ed il relativo significato sono riportati in tab. 7.6.

Valore	Significato
SA_NOCLDSTOP	Se il segnale è <code>SIGCHLD</code> allora non deve essere notificato quando il processo figlio viene fermato da uno dei segnali <code>SIGSTOP</code> , <code>SIGTSTP</code> , <code>SIGTTIN</code> o <code>SIGTTOU</code> , questo flag ha significato solo quando si imposta un gestore per <code>SIGCHLD</code> .
SA_NOCLDWAIT	Se il segnale è <code>SIGCHLD</code> e si richiede di ignorare il segnale con <code>SIG_IGN</code> allora i processi figli non diventano <i>zombie</i> quando terminano; questa funzionalità è stata introdotta nel kernel 2.6 e va a modificare il comportamento di <code>waitpid</code> come illustrato in sez. 3.1.5, se si installa un gestore con questo flag attivo il segnale <code>SIGCHLD</code> viene comunque generato.
SA_NODEFER	Evita che il segnale corrente sia bloccato durante l'esecuzione del gestore.
SA_NOMASK	Nome obsoleto e sinonimo non standard di <code>SA_NODEFER</code> , non deve essere più utilizzato.
SA_ONESHOT	Nome obsoleto e sinonimo non standard di <code>SA_RESETHAND</code> , non deve essere più utilizzato.
SA_ONSTACK	Stabilisce l'uso di uno <i>stack</i> alternativo per l'esecuzione del gestore (vedi sez. 7.5.3).
SA_RESETHAND	Ristabilisce l'azione per il segnale al valore predefinito una volta che il gestore è stato lanciato, riproduce cioè il comportamento della semantica inaffidabile.
SA_RESTART	Riavvia automaticamente le <i>slow system call</i> quando vengono interrotte dal suddetto segnale, riproduce cioè il comportamento standard di BSD.
SA_SIGINFO	Deve essere specificato quando si vuole usare un gestore in forma estesa usando <code>sa_sigaction</code> al posto di <code>sa_handler</code> .

Tabella 7.6: Valori del campo `sa_flag` della struttura `sigaction`.

Come si può notare in fig. 7.8 `sigaction` permette di utilizzare due forme diverse di gestore,⁹ da specificare, a seconda dell'uso o meno del flag `SA_SIGINFO`, rispettivamente attraverso i campi `sa_sigaction` o `sa_handler`. Quest'ultima è quella classica usata anche con `signal`, mentre la prima permette di usare un gestore più complesso, in grado di ricevere informazioni più dettagliate dal sistema, attraverso la struttura `siginfo_t`, riportata in fig. 7.9. I due cam-

⁹la possibilità è prevista dallo standard POSIX.1b, ed è stata aggiunta nei kernel della serie 2.1.x con l'introduzione dei segnali *real-time* (vedi sez. 7.5.1); in precedenza era possibile ottenere alcune informazioni addizionali usando `sa_handler` con un secondo parametro addizionale di tipo `sigcontext`, che adesso è deprecato.

pi devono essere usati in maniera alternativa, in certe implementazioni questi campi vengono addirittura definiti come una union.¹⁰

Installando un gestore di tipo `sa_sigaction` diventa allora possibile accedere alle informazioni restituite attraverso il puntatore a questa struttura. Tutti i segnali impostano i campi `si_signo`, che riporta il numero del segnale ricevuto, `si_errno`, che riporta, quando diverso da zero, il codice dell'errore associato al segnale, e `si_code`, che viene usato dal kernel per specificare maggiori dettagli riguardo l'evento che ha causato l'emissione del segnale.

```

siginfo_t {
    int      si_signo;    /* Signal number */
    int      si_errno;    /* An errno value */
    int      si_code;     /* Signal code */
    int      si_trapno;   /* Trap number that caused hardware-generated
                          signal (unused on most architectures) */

    pid_t    si_pid;     /* Sending process ID */
    uid_t    si_uid;     /* Real user ID of sending process */
    int      si_status;   /* Exit value or signal */
    clock_t  si_utime;    /* User time consumed */
    clock_t  si_stime;    /* System time consumed */
    sigval_t si_value;    /* Signal value */
    int      si_int;      /* POSIX.1b signal */
    void *   si_ptr;      /* POSIX.1b signal */
    int      si_overrun;  /* Timer overrun count; POSIX.1b timers */
    int      si_timerid;  /* Timer ID; POSIX.1b timers */
    void *   si_addr;     /* Memory location which caused fault */
    long     si_band;     /* Band event (was int before glibc 2.3.2) */
    int      si_fd;       /* File descriptor */
    short    si_addr_lsb; /* Least significant bit of address (since Linux 2.6.32) */
    void     *si_lower;   /* Lower bound when address violation occurred (since Linux 3.19) */
    void     *si_upper;   /* Upper bound when address violation occurred (since Linux 3.19) */
    int      si_pkey;     /* Protection key on PTE that caused fault (since Linux 4.6) */
    void     *si_call_addr; /* Address of system call instruction (since Linux 3.5) */
    int      si_syscall;  /* Number of attempted system call (since Linux 3.5) */
    unsigned int si_arch; /* Architecture of attempted system call
                          (since Linux 3.5) */
}

```

Figura 7.9: La struttura `siginfo_t`.

In generale `si_code` contiene, per i segnali generici, per quelli *real-time* e per tutti quelli inviati tramite da un processo con `kill` o affini, le informazioni circa l'origine del segnale stesso, ad esempio se generato dal kernel, da un timer, da `kill`, ecc. Il valore viene sempre espresso come una costante,¹¹ ed i valori possibili in questo caso sono riportati in tab. 7.7.

Nel caso di alcuni segnali però il valore di `si_code` viene usato per fornire una informazione specifica relativa alle motivazioni della ricezione dello stesso; ad esempio i vari segnali di errore (`SIGILL`, `SIGFPE`, `SIGSEGV` e `SIGBUS`) lo usano per fornire maggiori dettagli riguardo l'errore, come il tipo di errore aritmetico, di istruzione illecita o di violazione di memoria; mentre alcuni segnali di controllo (`SIGCHLD`, `SIGTRAP` e `SIGPOL`) forniscono altre informazioni specifiche.

In questo caso il valore del campo `si_code` deve essere verificato nei confronti delle diverse costanti previste per ciascuno di detti segnali; dato che si tratta di costanti, e non di una

¹⁰la direttiva `union` del linguaggio C definisce una variabile complessa, analoga a una struttura, i cui campi indicano i diversi tipi di valori che possono essere salvati, in maniera alternativa, all'interno della stessa.

¹¹le definizioni di tutti i valori possibili si trovano in `bits/siginfo.h`.

Valore	Significato
SI_USER	Generato da kill o raise o affini.
SI_KERNEL	Inviato direttamente dal kernel.
SI_QUEUE	Inviato con sigqueue (vedi sez. 7.5.1).
SI_TIMER	Scadenza di un <i>POSIX timer</i> (vedi sez. 7.5.2).
SI_MSGQ	Inviato al cambiamento di stato di una coda di messaggi POSIX (vedi sez. 11.4.2), introdotto con il kernel 2.6.6.
SI_ASYNCIO	Una operazione di I/O asincrono (vedi sez. 10.3.3) è stata completata.
SI_SIGIO	Segnale di SIGIO da una coda (vedi sez. 10.3).
SI_TKILL	Inviato da tkill o tkill (vedi sez. ??), introdotto con il kernel 2.4.19.

Tabella 7.7: Valori del campo `si_code` della struttura `sigaction` per i segnali generici.

maschera binaria, i valori numerici vengono riutilizzati e ciascuno di essi avrà un significato diverso a seconda del segnale a cui è associato.

L'elenco dettagliato dei nomi di queste costanti è riportato nelle diverse sezioni di tab. 7.8 che sono state ordinate nella sequenza in cui si sono appena citati i rispettivi segnali, il prefisso del nome indica comunque in maniera diretta il segnale a cui le costanti fanno riferimento.

Valore	Significato
ILL_ILLOPC	Codice di operazione illegale.
ILL_ILLOPN	Operando illegale.
ILL_ILLADR	Modo di indirizzamento illegale.
ILL_ILLTRP	Trappola di processore illegale.
ILL_PRVOPC	Codice di operazione privilegiato.
ILL_PRIVREG	Registro privilegiato.
ILL_COPROC	Errore del coprocessore.
ILL_BADSTK	Errore nello stack interno.
FPE_INTDIV	Divisione per zero intera.
FPE_INTOVF	Overflow intero.
FPE_FLTDIV	Divisione per zero in virgola mobile.
FPE_FLTOVF	Overflow in virgola mobile.
FPE_FLTUND	Underflow in virgola mobile.
FPE_FLTRES	Risultato in virgola mobile non esatto.
FPE_FLTINV	Operazione in virgola mobile non valida.
FPE_FLTSUB	Mantissa? fuori intervallo.
SEGV_MAPERR	Indirizzo non mappato.
SEGV_ACCERR	Permessi non validi per l'indirizzo.
BUS_ADRALN	Allineamento dell'indirizzo non valido.
BUS_ADRERR	Indirizzo fisico inesistente.
BUS_OBJERR	Errore hardware sull'indirizzo.
TRAP_BRKPT	Breakpoint sul processo.
TRAP_TRACE	Trappola di tracciamento del processo.
CLD_EXITED	Il figlio è uscito.
CLD_KILLED	Il figlio è stato terminato.
CLD_DUMPED	Il figlio è terminato in modo anormale.
CLD_TRAPPED	Un figlio tracciato ha raggiunto una trappola.
CLD_STOPPED	Il figlio è stato fermato.
CLD_CONTINUED	Il figlio è ripartito.
POLL_IN	Disponibili dati in ingresso.
POLL_OUT	Spazio disponibile sul buffer di uscita.
POLL_MSG	Disponibili messaggi in ingresso.
POLL_ERR	Errore di I/O.
POLL_PRI	Disponibili dati di alta priorità in ingresso.
POLL_HUP	Il dispositivo è stato disconnesso.

Tabella 7.8: Valori del campo `si_code` della struttura `sigaction` impostati rispettivamente dai segnali SIGILL, SIGFPE, SIGSEGV, SIGBUS, SIGCHLD, SIGTRAP e SIGPOLL/SIGIO.

Il resto della struttura `siginfo_t` è definito come una `union` ed i valori eventualmente presenti dipendono dal segnale ricevuto, così `SIGCHLD` ed i segnali *real-time* (vedi sez. 7.5.1) inviati tramite `kill` avvalorano `si_pid` e `si_uid` coi valori corrispondenti al processo che ha emesso il segnale, `SIGCHLD` avvalora anche i campi `si_status`, `si_utime` e `si_stime` che indicano rispettivamente lo stato di uscita, l'*user time* e il *system time* (vedi sez. 6.4.2) usati dal processo; `SIGILL`, `SIGFPE`, `SIGSEGV` e `SIGBUS` avvalorano `si_addr` con l'indirizzo in cui è avvenuto l'errore, `SIGIO` (vedi sez. 10.3.3) avvalora `si_fd` con il numero del file descriptor e `si_band` per i dati urgenti (vedi sez. 18.1.4) su un socket, il segnale inviato alla scadenza di un POSIX timer (vedi sez. 7.5.2) avvalora i campi `si_timerid` e `si_overshoot`.

Benché sia possibile usare nello stesso programma sia `sigaction` che `signal` occorre molta attenzione, in quanto le due funzioni possono interagire in maniera anomala. Infatti l'azione specificata con `sigaction` contiene un maggior numero di informazioni rispetto al semplice indirizzo del gestore restituito da `signal`. Per questo motivo se si usa quest'ultima per installare un gestore sostituendone uno precedentemente installato con `sigaction`, non sarà possibile effettuare un ripristino corretto dello stesso.

Per questo è sempre opportuno usare `sigaction`, che è in grado di ripristinare correttamente un gestore precedente, anche se questo è stato installato con `signal`. In generale poi non è il caso di usare il valore di ritorno di `signal` come campo `sa_handler`, o viceversa, dato che in certi sistemi questi possono essere diversi. In definitiva dunque, a meno che non si sia vincolati all'aderenza stretta allo standard ISO C, è sempre il caso di evitare l'uso di `signal` a favore di `sigaction`.

```

1 typedef void SigFunc(int);
2 inline SigFunc * Signal(int signo, SigFunc *func)
3 {
4     struct sigaction new_handl, old_handl;
5     new_handl.sa_handler = func;
6     /* clear signal mask: no signal blocked during execution of func */
7     if (sigemptyset(&new_handl.sa_mask)!=0){ /* initialize signal set */
8         return SIG_ERR;
9     }
10    new_handl.sa_flags=0; /* init to 0 all flags */
11    /* change action for signo signal */
12    if (sigaction(signo, &new_handl, &old_handl)){
13        return SIG_ERR;
14    }
15    return (old_handl.sa_handler);
16 }

```

Figura 7.10: La funzione `Signal`, equivalente a `signal`, definita attraverso `sigaction`.

Per questo motivo si è provveduto, per mantenere un'interfaccia semplificata che abbia le stesse caratteristiche di `signal`, a definire attraverso `sigaction` una funzione equivalente `Signal`, il cui codice è riportato in fig. 7.10 (il codice completo si trova nel file `SigHand.c` nei sorgenti allegati). Anche in questo caso, per semplificare la definizione si è poi definito un apposito tipo `SigFunc` per esprimere in modo più comprensibile la forma di un gestore di segnale.

Si noti come, essendo la funzione estremamente semplice, essa è definita come `inline`. Questa direttiva viene usata per dire al compilatore di trattare la funzione cui essa fa riferimento in maniera speciale inserendo il codice direttamente nel testo del programma. Anche se i compilatori più moderni sono in grado di effettuare da soli queste manipolazioni (impo-

stando le opportune ottimizzazioni) questa è una tecnica usata per migliorare le prestazioni per le funzioni piccole ed usate di frequente, in particolare nel kernel, dove in certi casi le ottimizzazioni dal compilatore, tarate per l'uso in *user space*, non sono sempre adatte.

In tal caso infatti le istruzioni per creare un nuovo frame nello *stack* per chiamare la funzione costituirebbero una parte rilevante del codice, appesantendo inutilmente il programma. Originariamente questo comportamento veniva ottenuto con delle macro, ma queste hanno tutta una serie di problemi di sintassi nel passaggio degli argomenti (si veda ad esempio [?]) che in questo modo possono essere evitati.

7.4.4 La gestione della maschera dei segnali o *signal mask*

Come spiegato in sez. 7.1.2 tutti i moderni sistemi unix-like permettono di bloccare temporaneamente (o di eliminare completamente, impostando come azione SIG_IGN) la consegna dei segnali ad un processo. Questo è fatto specificando la cosiddetta *maschera dei segnali* (o *signal mask*) del processo¹² cioè l'insieme dei segnali la cui consegna è bloccata.

Abbiamo accennato in sez. 3.1.3 che la maschera dei segnali viene ereditata dal padre alla creazione di un processo figlio, e abbiamo visto al paragrafo precedente che essa può essere modificata durante l'esecuzione di un gestore ed automaticamente ripristinata quando questo ritorna, attraverso l'uso dal campo `sa_mask` di `sigaction`.

Uno dei problemi evidenziatisi con l'esempio di fig. 7.7 è che in molti casi è necessario proteggere delle sezioni di codice, in modo da essere sicuri che essi siano eseguite senza interruzioni da parte di un segnale. Nel caso in questione si trattava della sezione di codice fra il controllo e la eventuale cancellazione del flag impostato dal gestore di un segnale che testimoniava l'avvenuta occorrenza dello stesso.

Come illustrato in sez. 3.4.1 le operazioni più semplici, come l'assegnazione o il controllo di una variabile, di norma sono atomiche, e qualora si voglia essere sicuri si può usare il tipo `sig_atomic_t`. Ma quando si devono eseguire più operazioni su delle variabili (nell'esempio citato un controllo ed una assegnazione) o comunque eseguire una serie di istruzioni, l'atomicità non è più possibile.

In questo caso, se si vuole essere sicuri di non poter essere interrotti da un segnale durante l'esecuzione di una sezione di codice, lo si può bloccare esplicitamente modificando la maschera dei segnali del processo con la funzione di sistema `sigprocmask`, il cui prototipo è:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
    Imposta la maschera dei segnali del processo corrente.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT si sono specificati indirizzi non validi.

EINVAL si è specificato un numero di segnale invalido.

La funzione usa l'insieme di segnali posto all'indirizzo passato nell'argomento `set` per modificare la maschera dei segnali del processo corrente. La modifica viene effettuata a seconda del valore dell'argomento `how`, secondo le modalità specificate in tab. 7.9. Qualora si specifichi un valore non nullo per `oldset` la maschera dei segnali corrente viene salvata a quell'indirizzo.

In questo modo diventa possibile proteggere delle sezioni di codice bloccando l'insieme di segnali voluto per poi riabilitarli alla fine della sezione critica. La funzione permette di risolvere problemi come quelli mostrati in fig. 7.7, proteggendo la sezione fra il controllo del flag e la sua cancellazione. La funzione può essere usata anche all'interno di un gestore, ad

¹²nel caso di Linux essa è mantenuta dal campo `blocked` della `task_struct` del processo.

Valore	Significato
SIG_BLOCK	L'insieme dei segnali bloccati è l'unione fra quello specificato e quello corrente.
SIG_UNBLOCK	I segnali specificati in set sono rimossi dalla maschera dei segnali, specificare la cancellazione di un segnale non bloccato è legale.
SIG_SETMASK	La maschera dei segnali è impostata al valore specificato da set.

Tabella 7.9: Valori e significato dell'argomento how della funzione sigprocmask.

esempio per riabilitare la consegna del segnale che l'ha invocato, in questo caso però occorre ricordare che qualunque modifica alla maschera dei segnali viene perduta al ritorno dallo stesso.

Benché con l'uso di `sigprocmask` si possano risolvere la maggior parte dei casi di *race condition* restano aperte alcune possibilità legate all'uso di `pause`. Il caso è simile a quello del problema illustrato nell'esempio di fig. 7.6, e cioè la possibilità che il processo riceva il segnale che si intende usare per uscire dallo stato di attesa invocato con `pause` immediatamente prima dell'esecuzione di quest'ultima. Per poter effettuare atomicamente la modifica della maschera dei segnali (di solito attivandone uno specifico) insieme alla sospensione del processo lo standard POSIX ha previsto la funzione di sistema `sigsuspend`, il cui prototipo è:

```
#include <signal.h>
int sigsuspend(const sigset_t *mask)
```

Imposta la maschera dei segnali mettendo in attesa il processo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT si sono specificati indirizzi non validi.

EINVAL si è specificato un numero di segnale invalido.

Come esempio dell'uso di queste funzioni proviamo a riscrivere un'altra volta l'esempio di implementazione di `sleep`. Abbiamo accennato in sez. 7.4.3 come con `sigaction` sia possibile bloccare `SIGALRM` nell'installazione dei gestori degli altri segnali, per poter usare l'implementazione vista in fig. 7.6 senza interferenze. Questo però comporta una precauzione ulteriore al semplice uso della funzione, vediamo allora come usando la nuova interfaccia è possibile ottenere un'implementazione, riportata in fig. 7.11 che non presenta neanche questa necessità.

Per evitare i problemi di interferenza con gli altri segnali in questo caso non si è usato l'approccio di fig. 7.6 evitando l'uso di `longjmp`. Come in precedenza il gestore (27-30) non esegue nessuna operazione, limitandosi a ritornare per interrompere il programma messo in attesa.

La prima parte della funzione (6-10) provvede ad installare l'opportuno gestore per `SIGALRM`, salvando quello originario, che sarà ripristinato alla conclusione della stessa (23); il passo successivo è quello di bloccare `SIGALRM` (11-14) per evitare che esso possa essere ricevuto dal processo fra l'esecuzione di `alarm` (16) e la sospensione dello stesso. Nel fare questo si salva la maschera corrente dei segnali, che sarà ripristinata alla fine (22), e al contempo si prepara la maschera dei segnali `sleep_mask` per riattivare `SIGALRM` all'esecuzione di `sigsuspend`.

In questo modo non sono più possibili *race condition* dato che `SIGALRM` viene disabilitato con `sigprocmask` fino alla chiamata di `sigsuspend`. Questo metodo è assolutamente generale e può essere applicato a qualunque altra situazione in cui si deve attendere per un segnale, i passi sono sempre i seguenti:

1. leggere la maschera dei segnali corrente e bloccare il segnale voluto con `sigprocmask`;

```

1 void alarm_hand(int);
2 unsigned int sleep(unsigned int seconds)
3 {
4     struct sigaction new_action, old_action;
5     sigset_t old_mask, stop_mask, sleep_mask;
6     /* set the signal handler */
7     sigemptyset(&new_action.sa_mask);           /* no signal blocked */
8     new_action.sa_handler = alarm_hand;         /* set handler */
9     new_action.sa_flags = 0;                   /* no flags */
10    sigaction(SIGALRM, &new_action, &old_action); /* install action */
11    /* block SIGALRM to avoid race conditions */
12    sigemptyset(&stop_mask);                     /* init mask to empty */
13    sigaddset(&stop_mask, SIGALRM);              /* add SIGALRM */
14    sigprocmask(SIG_BLOCK, &stop_mask, &old_mask); /* add SIGALRM to blocked */
15    /* send the alarm */
16    alarm(seconds);
17    /* going to sleep enabling SIGALRM */
18    sleep_mask = old_mask;                       /* take mask */
19    sigdelset(&sleep_mask, SIGALRM);             /* remove SIGALRM */
20    sigsuspend(&sleep_mask);                     /* go to sleep */
21    /* restore previous settings */
22    sigprocmask(SIG_SETMASK, &old_mask, NULL);   /* reset signal mask */
23    sigaction(SIGALRM, &old_action, NULL);       /* reset signal action */
24    /* return remaining time */
25    return alarm(0);
26 }
27 void alarm_hand(int sig)
28 {
29     return;    /* just return to interrupt sigsuspend */
30 }

```

Figura 7.11: Una implementazione completa di sleep.

2. mandare il processo in attesa con `sigsuspend` abilitando la ricezione del segnale voluto;
3. ripristinare la maschera dei segnali originaria.

Per quanto possa sembrare strano bloccare la ricezione di un segnale per poi riabilitarla immediatamente dopo, in questo modo si evita il *deadlock* dovuto all'arrivo del segnale prima dell'esecuzione di `sigsuspend`.

7.4.5 Criteri di programmazione per i gestori dei segnali

Abbiamo finora parlato dei gestori dei segnali come funzioni chiamate in corrispondenza della consegna di un segnale. In realtà un gestore non può essere una funzione qualunque, in quanto esso può essere eseguito in corrispondenza all'interruzione in un punto qualunque del programma principale, cosa che ad esempio può rendere problematico chiamare all'interno di un gestore di segnali la stessa funzione che dal segnale è stata interrotta.

Il concetto è comunque più generale e porta ad una distinzione fra quelle che POSIX chiama *funzioni insicure* (*signal unsafe function*) e *funzioni sicure* (o più precisamente *signal safe function*). Quando un segnale interrompe una funzione insicura ed il gestore chiama al suo interno una funzione insicura il sistema può dare luogo ad un comportamento indefinito, la cosa non avviene invece per le funzioni sicure.

Tutto questo significa che la funzione che si usa come gestore di segnale deve essere programmata con molta cura per evitare questa evenienza e che non è possibile utilizzare al suo interno una qualunque funzione di sistema, se si vogliono evitare questi problemi si può ricorrere soltanto all'uso delle funzioni considerate sicure.

L'elenco delle funzioni considerate sicure varia a seconda della implementazione utilizzata e dello standard a cui si fa riferimento. Non è riportata una lista specifica delle funzioni sicure per Linux, e si suppone pertanto che siano quelle richieste dallo standard. Secondo quanto richiesto dallo standard POSIX 1003.1 nella revisione del 2003, le "*signal safe function*" che possono essere chiamate anche all'interno di un gestore di segnali sono tutte quelle della lista riportata in fig. 7.12.

```
_exit, abort, accept, access, aio_error aio_return, aio_suspend, alarm, bind, cfgetispeed, cfgetospeed,
cfsetispeed, cfsetospeed, chdir, chmod, chown, clock_gettime, close, connect, creat, dup, dup2, execl,
execve, fchmod, fchown, fcntl, fdatsync, fork, fpathconf, fstat, fsync, ftruncate, getegid, geteuid, getgid,
getgroups, getpeername, getpgrp, getpid, getppid, getsockname, getsockopt, getuid, kill, link, listen,
lseek, lstat, mkdir, mkfifo, open, pathconf, pause, pipe, poll, posix_trace_event, pselect, raise, read,
readlink, recv, recvfrom, recvmsg, rename, rmdir, select, sem_post, send, sendmsg, sendto, setgid, setpgid,
setuid, setsockopt, setuid, shutdown, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset, sigismember,
signal, sigpause, sigpending, sigprocmask, sigqueue, sigset, sigsuspend, sleep, socket, socketpair, stat,
symlink, sysconf, tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsendbreak, tcsetattr, tcsetpgrp, time,
timer_getoverrun, timer_gettime, timer_settime, times, umask, unlink, utime, wait, waitpid, write.
```

Figura 7.12: Elenco delle funzioni sicure secondo lo standard POSIX 1003.1-2003.

Lo standard POSIX.1-2004 modifica la lista di fig. 7.12 aggiungendo le funzioni `_Exit` e `socketpair`, mentre lo standard POSIX.1-2008 rimuove dalla lista le tre funzioni `fpathconf`, `pathconf`, `sysconf` e vi aggiunge le ulteriori funzioni in fig. 7.13.

```
execl, execv, faccessat, fchmodat, fchownat, fexecve, fstatat, futimens, linkat, mkdirat, mkfifoat, mknod,
mknodat, openat, readlinkat, renameat, symlinkat, unlinkat, utimensat, utimes.
```

Figura 7.13: Ulteriori funzioni sicure secondo lo standard POSIX.1-2008.

Per questo motivo è opportuno mantenere al minimo indispensabile le operazioni effettuate all'interno di un gestore di segnali, qualora si debbano compiere operazioni complesse è sempre

preferibile utilizzare la tecnica in cui si usa il gestore per impostare il valore di una qualche variabile globale, e poi si eseguono le operazioni complesse nel programma verificando (con tutti gli accorgimenti visti in precedenza) il valore di questa variabile tutte le volte che si è rilevata una interruzione dovuta ad un segnale.

7.5 Funzionalità avanzate

Tratteremo in questa ultima sezione alcune funzionalità avanzate relativa ai segnali ed in generale ai meccanismi di notifica, a partire dalla funzioni introdotte per la gestione dei cosiddetti “*segnali real-time*”, alla gestione avanzata delle temporizzazioni e le nuove interfacce per la gestione di segnali ed eventi attraverso l’uso di file descriptor.

7.5.1 I segnali *real-time*

Lo standard POSIX.1b, nel definire una serie di nuove interfacce per i servizi *real-time*, ha introdotto una estensione del modello classico dei segnali che presenta dei significativi miglioramenti,¹³ in particolare sono stati superati tre limiti fondamentali dei segnali classici:

I segnali non sono accumulati

se più segnali vengono generati prima dell’esecuzione di un gestore questo sarà eseguito una sola volta, ed il processo non sarà in grado di accorgersi di quante volte l’evento che ha generato il segnale è accaduto.

I segnali non trasportano informazione

i segnali classici non prevedono altra informazione sull’evento che li ha generati se non il fatto che sono stati emessi (tutta l’informazione che il kernel associa ad un segnale è il suo numero).

I segnali non hanno un ordine di consegna

l’ordine in cui diversi segnali vengono consegnati è casuale e non prevedibile. Non è possibile stabilire una priorità per cui la reazione a certi segnali ha la precedenza rispetto ad altri.

Per poter superare queste limitazioni lo standard POSIX.1b ha introdotto delle nuove caratteristiche, che sono state associate ad una nuova classe di segnali, che vengono chiamati *segnali real-time*, in particolare le funzionalità aggiunte sono:

1. i segnali sono inseriti in una coda che permette di consegnare istanze multiple dello stesso segnale qualora esso venga inviato più volte prima dell’esecuzione del gestore; si assicura così che il processo riceva un segnale per ogni occorrenza dell’evento che lo genera;
2. è stata introdotta una priorità nella consegna dei segnali: i segnali vengono consegnati in ordine a seconda del loro valore, partendo da quelli con un numero minore, che pertanto hanno una priorità maggiore;
3. è stata introdotta la possibilità di restituire dei dati al gestore, attraverso l’uso di un apposito campo `si_value` nella struttura `siginfo_t`, accessibile tramite gestori di tipo `sa_sigaction`.

¹³questa estensione è stata introdotta in Linux a partire dal kernel 2.1.43, e dalla versione 2.1 della *glibc*.

Tutte queste nuove funzionalità eccetto l'ultima, che, come illustrato in sez. 7.4.3, è disponibile anche con i segnali ordinari, si applicano solo ai nuovi segnali *real-time*; questi ultimi sono accessibili in un intervallo di valori specificati dalle due costanti SIGRTMIN e SIGRTMAX, che specificano il numero minimo e massimo associato ad un segnale *real-time*.

Su Linux di solito il primo valore è 33, mentre il secondo è _NSIG-1, che di norma (vale a dire sulla piattaforma i386) è 64. Questo dà un totale di 32 segnali disponibili, contro gli almeno 8 richiesti da POSIX.1b. Si tenga presente però che i primi segnali *real-time* disponibili vengono usati dalla *glibc* per l'implementazione dei *thread* POSIX (vedi sez. 12.3), ed il valore di SIGRTMIN viene modificato di conseguenza.¹⁴

Per questo motivo nei programmi che usano i segnali *real-time* non si deve mai usare un valore assoluto dato che si correrebbe il rischio di utilizzare un segnale in uso alle librerie, ed il numero del segnale deve invece essere sempre specificato in forma relativa a SIGRTMIN (come SIGRTMIN + n) avendo inoltre cura di controllare di non aver mai superato SIGRTMAX.

I segnali con un numero più basso hanno una priorità maggiore e vengono consegnati per primi, inoltre i segnali *real-time* non possono interrompere l'esecuzione di un gestore di un segnale a priorità più alta; la loro azione predefinita è quella di terminare il programma. I segnali ordinari hanno tutti la stessa priorità, che è più alta di quella di qualunque segnale *real-time*. Lo standard non definisce niente al riguardo ma Linux, come molte altre implementazioni, adotta questa politica.

Si tenga presente che questi nuovi segnali non sono associati a nessun evento specifico, a meno di non richiedere specificamente il loro utilizzo in meccanismi di notifica come quelli per l'I/O asincrono (vedi sez. 10.3.3) o per le code di messaggi POSIX (vedi sez. 11.4.2), pertanto devono essere inviati esplicitamente.

Inoltre, per poter usufruire della capacità di restituire dei dati, i relativi gestori devono essere installati con `sigaction`, specificando per `sa_flags` la modalità SA_SIGINFO che permette di utilizzare la forma estesa `sa_sigaction` del gestore (vedi sez. 7.4.3). In questo modo tutti i segnali *real-time* possono restituire al gestore una serie di informazioni aggiuntive attraverso l'argomento `siginfo_t`, la cui definizione è stata già vista in fig. 7.9, nella trattazione dei gestori in forma estesa.

In particolare i campi utilizzati dai segnali *real-time* sono `si_pid` e `si_uid` in cui vengono memorizzati rispettivamente il *PID* e l'*UID* effettivo del processo che ha inviato il segnale, mentre per la restituzione dei dati viene usato il campo `si_value`.

```
typedef union signal {
    int sival_int;
    void *sival_ptr;
} sigval_t;
```

Figura 7.14: La definizione dell'unione `signal`, definita anche come tipo `sigval_t`.

Detto campo, identificato con il tipo di dato `sigval_t`, è una *union* di tipo `signal` (la sua definizione è in fig. 7.14) in cui può essere memorizzato o un valore numerico, se usata nella forma `sival_int`, o un puntatore, se usata nella forma `sival_ptr`. L'unione viene usata dai segnali *real-time* e da vari meccanismi di notifica per restituire dati al gestore del segnale in `si_value`. Un campo di tipo `sigval_t` è presente anche nella struttura `sigevent` (definita

¹⁴per la precisione vengono usati i primi tre per la vecchia implementazione dei *LinuxThread* ed i primi due per la nuova NTPL (*New Thread Posix Library*), il che comporta che SIGRTMIN a seconda dei casi può assumere i valori 34 o 35.

in fig. 7.15) che viene usata dai meccanismi di notifica come quelli per i timer POSIX (vedi sez. 7.5.2), l'I/O asincrono (vedi sez. 10.3.3) o le code di messaggi POSIX (vedi sez. 11.4.2).

A causa delle loro caratteristiche, la funzione `kill` non è adatta ad inviare segnali *real-time*, poiché non è in grado di fornire alcun valore per il campo `si_value` restituito nella struttura `siginfo_t` prevista da un gestore in forma estesa. Per questo motivo lo standard ha previsto una nuova funzione, `sigqueue`, il cui prototipo è:

```
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value)
```

Invia un segnale con un valore di informazione.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN la coda è esaurita, ci sono già SIGQUEUE_MAX segnali in attesa si consegna.

EINVAL si è specificato un valore non valido per `signo`.

EPERM non si hanno privilegi appropriati per inviare il segnale al processo specificato.

ESRCH il processo `pid` non esiste.

La funzione invia il segnale indicato dall'argomento `signo` al processo indicato dall'argomento `pid`. Per il resto il comportamento della funzione è analogo a quello di `kill`, ed i privilegi occorrenti ad inviare il segnale ad un determinato processo sono gli stessi; un valore nullo di `signo` permette di verificare le condizioni di errore senza inviare nessun segnale.

Se il segnale è bloccato la funzione ritorna immediatamente, se si è installato un gestore con `SA_SIGINFO` e ci sono risorse disponibili, (vale a dire che c'è posto nella coda dei segnali *real-time*) esso viene inserito e diventa pendente. Una volta consegnato il segnale il gestore otterrà nel campo `si_code` di `siginfo_t` il valore `SI_QUEUE` e nel campo `si_value` il valore indicato nell'argomento `value`. Se invece si è installato un gestore nella forma classica il segnale sarà generato, ma tutte le caratteristiche tipiche dei segnali *real-time* (priorità e coda) saranno perse.

Per lo standard POSIX la profondità della coda è indicata dalla costante `SIGQUEUE_MAX`, una delle tante costanti di sistema definite dallo standard POSIX che non abbiamo riportato esplicitamente in sez. 6.1.1. Il suo valore minimo secondo lo standard, `_POSIX_SIGQUEUE_MAX`, è pari a 32. Nel caso di Linux la coda ha una dimensione variabile; fino alla versione 2.6.7 c'era un limite massimo globale che poteva essere impostato come parametro del kernel in `/proc/sys/kernel/rtsig-max` ed il valore predefinito era pari a 1024. A partire dal kernel 2.6.8 il valore globale è stato rimosso e sostituito dalla risorsa `RLIMIT_SIGPENDING` associata al singolo utente, che può essere modificata con `setrlimit` come illustrato in sez. 6.3.2.

Lo standard POSIX.1b definisce inoltre delle nuove funzioni di sistema che permettono di gestire l'attesa di segnali specifici su una coda, esse servono in particolar modo nel caso dei *thread*, in cui si possono usare i segnali *real-time* come meccanismi di comunicazione elementare; la prima di queste è `sigwait`, il cui prototipo è:

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig)
```

Attende la ricezione di un segnale.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINTR la funzione è stata interrotta.

EINVAL si è specificato un valore non valido per
ed inoltre EFAULT nel suo significato generico.

La funzione estrae dall'insieme dei segnali pendenti uno qualunque fra quelli indicati nella *signal set* specificato in `set`, il cui valore viene restituito nella variabile puntata da `sig`. Se

sono pendenti più segnali, viene estratto quello a priorità più alta, cioè quello con il numero più basso. Se, nel caso di segnali *real-time*, c'è più di un segnale pendente, ne verrà estratto solo uno. Una volta estratto il segnale non verrà più consegnato, e se era in una coda il suo posto sarà liberato. Se non c'è nessun segnale pendente il processo viene bloccato fintanto che non ne arriva uno.

Per un funzionamento corretto la funzione richiede che alla sua chiamata i segnali di **set** siano bloccati. In caso contrario si avrebbe un conflitto con gli eventuali gestori: pertanto non si deve utilizzare per lo stesso segnale questa funzione e **sigaction**. Se questo non avviene il comportamento del sistema è indeterminato: il segnale può sia essere consegnato che essere ricevuto da **sigwait**, il tutto in maniera non prevedibile.

Lo standard POSIX.1b definisce altre due funzioni di sistema, anch'esse usate prevalentemente con i *thread*; **sigwaitinfo** e **sigtimedwait**, i relativi prototipi sono:

```
#include <signal.h>
int sigwaitinfo(const sigset_t *set, siginfo_t *info)
    Attende un segnale con le relative informazioni.
int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)
    Attende un segnale con le relative informazioni per un tempo massimo.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso **errno** assumerà uno gli stessi valori di **sigwait** ai quali si aggiunge per **sigtimedwait**:

EAGAIN si è superato il timeout senza che un segnale atteso sia stato ricevuto.

Entrambe le funzioni sono estensioni di **sigwait**. La prima permette di ricevere, oltre al numero del segnale, anche le informazioni ad esso associate tramite l'argomento **info**; in particolare viene restituito il numero del segnale nel campo **si_signo**, la sua causa in **si_code**, e se il segnale è stato immesso sulla coda con **sigqueue**, il valore di ritorno ad esso associato viene riportato in **si_value**, che altrimenti è indefinito.

La seconda è identica alla prima ma in più permette di specificare un timeout con l'argomento omonimo, scaduto il quale ritornerà con un errore. Se si specifica per **timeout** un puntatore nullo il comportamento sarà identico a **sigwaitinfo**. Se si specifica un tempo di timeout nullo e non ci sono segnali pendenti la funzione ritornerà immediatamente, in questo modo si può eliminare un segnale dalla coda senza dover essere bloccati qualora esso non sia presente.

L'uso di queste funzioni è principalmente associato alla gestione dei segnali con i *thread*. In genere esse vengono chiamate dal *thread* incaricato della gestione, che al ritorno della funzione esegue il codice che usualmente sarebbe messo nel gestore, per poi ripetere la chiamata per mettersi in attesa del segnale successivo. Questo ovviamente comporta che non devono essere installati gestori, che solo il *thread* di gestione deve usare **sigwait** e che i segnali gestiti in questa maniera, per evitare che venga eseguita l'azione predefinita, devono essere mascherati per tutti i *thread*, compreso quello dedicato alla gestione, che potrebbe riceverlo fra due chiamate successive.

7.5.2 La gestione avanzata delle temporizzazioni

Sia le funzioni per la gestione dei tempi viste in sez. 6.4.2 che quelle per la gestione dei timer di sez. 7.3.4 sono state a lungo limitate dalla risoluzione massima dei tempi dell'orologio interno del kernel, che era quella ottenibile dal timer di sistema che governa lo *scheduler*, e quindi limitate dalla frequenza dello stesso che si ricordi, come già illustrato in sez. 3.1.1, è data dal valore della costante **HZ**.

I contatori usati per il calcolo dei tempi infatti erano basati sul numero di *jiffies* che vengono incrementati ad ogni *clock tick* del timer di sistema, il che comportava anche, come

accennato in sez. 7.3.4 per `setitimer`, problemi per il massimo periodo di tempo copribile da alcuni di questi orologi, come quelli associati al *process time* almeno fino a quando, con il kernel 2.6.16, non è stato rimosso il limite di un valore a 32 bit per i *jiffies*.

Nelle architetture moderne però tutti i computer sono dotati di temporizzatori hardware che possono supportare risoluzioni molto elevate, ed in maniera del tutto indipendente dalla frequenza scelta per il timer di sistema che governa lo *scheduler*, normalmente si possono ottenere precisioni fino al microsecondo, andando molto oltre in caso di hardware dedicato.

Per questo lo standard POSIX.1-2001 ha previsto una serie di nuove funzioni relative a quelli che vengono chiamati “*orologi real-time*”, in grado di supportare risoluzioni fino al nanosecondo. Inoltre le CPU più moderne sono dotate a loro volta di contatori ad alta definizione che consentono una grande accuratezza nella misura del tempo da esse dedicato all'esecuzione di un processo.

Per usare queste funzionalità ed ottenere risoluzioni temporali più accurate, occorre però un opportuno supporto da parte del kernel, ed i cosiddetti *high resolution timer* che consentono di fare ciò sono stati introdotti nel kernel ufficiale solo a partire dalla versione 2.6.21.¹⁵ Le funzioni definite dallo standard POSIX per gestire orologi ad alta definizione però erano già presenti, essendo stata introdotta insieme ad altre funzioni per il supporto delle estensioni *real-time* con il rilascio del kernel 2.6, ma la risoluzione effettiva era nominale.

A tutte le implementazioni che si rifanno a queste estensioni è richiesto di disporre di una versione *real-time* almeno per l'orologio generale di sistema, quello che mantiene il *calendar time* (vedi sez. 6.4.3), che in questa forma deve indicare il numero di secondi e nanosecondi passati a partire dal primo gennaio 1970 (*The Epoch*). Si ricordi infatti che l'orologio ordinario usato dal *calendar time* riporta solo un numero di secondi, e che la risoluzione effettiva normalmente non raggiunge il nanosecondo (a meno di hardware specializzato). Oltre all'orologio generale di sistema possono essere presenti altri tipi di orologi *real-time*, ciascuno dei quali viene identificato da un opportuno valore di una variabile di tipo `clockid_t`; un elenco di quelli disponibili su Linux è riportato in tab. 7.10.

Per poter utilizzare queste funzionalità la *glibc* richiede che la macro `_POSIX_C_SOURCE` sia definita ad un valore maggiore o uguale di 199309L (vedi sez. 1.3.7), inoltre i programmi che le usano devono essere collegati con la libreria delle estensioni *real-time* usando esplicitamente l'opzione `-lrt`.

Si tenga presente inoltre che la disponibilità di queste funzionalità avanzate può essere controllato dalla definizione della macro `_POSIX_TIMERS` ad un valore maggiore di 0, e che le ulteriori macro `_POSIX_MONOTONIC_CLOCK`, `_POSIX_CPUTIME` e `_POSIX_THREAD_CPUTIME` indicano la presenza dei rispettivi orologi di tipo `CLOCK_MONOTONIC`, `CLOCK_PROCESS_CPUTIME_ID` e `CLOCK_THREAD_CPUTIME_ID`; tutte queste macro sono definite in `unistd.h`, che pertanto deve essere incluso per poterle controllare. Infine se il kernel ha il supporto per gli *high resolution timer* un elenco degli orologi e dei timer può essere ottenuto tramite il file `/proc/timer_list`.

Le due funzioni che ci consentono rispettivamente di modificare o leggere il valore per uno degli orologi *real-time* sono `clock_settime` e `clock_gettime`; i rispettivi prototipi sono:

```
#include <time.h>
int clock_settime(clockid_t clockid, const struct timespec *tp)
                                                    Imposta un orologio real-time.
int clock_gettime(clockid_t clockid, struct timespec *tp)
                                                    Legge un orologio real-time.
```

¹⁵per il supporto deve essere stata abilitata l'opzione di compilazione `CONFIG_HIGH_RES_TIMERS`, il supporto era però disponibile anche in precedenza nei patch facenti parte dello sviluppo delle estensioni *real-time* del kernel, per cui alcune distribuzioni possono averlo anche con versioni precedenti del kernel.

Valore	Significato
CLOCK_REALTIME	Orologio <i>real-time</i> di sistema, può essere impostato solo con privilegi amministrativi.
CLOCK_MONOTONIC	Orologio che indica un tempo monotono crescente (a partire da un tempo iniziale non specificato) che non può essere modificato e non cambia neanche in caso di reimpostazione dell'orologio di sistema.
CLOCK_PROCESS_CPUTIME_ID	Contatore del tempo di CPU usato da un processo (il <i>process time</i> di sez. 6.4.2, nel totale di <i>system time</i> e <i>user time</i>) comprensivo di tutto il tempo di CPU usato da eventuali <i>thread</i> .
CLOCK_THREAD_CPUTIME_ID	Contatore del tempo di CPU (<i>user time</i> e <i>system time</i>) usato da un singolo <i>thread</i> .
CLOCK_MONOTONIC_RAW	Simile al precedente, ma non subisce gli aggiustamenti dovuti all'uso di NTP (viene usato per fare riferimento ad una fonte hardware). Questo orologio è specifico di Linux, ed è disponibile a partire dal kernel 2.6.28.
CLOCK_BOOTTIME	Identico a CLOCK_MONOTONIC ma tiene conto anche del tempo durante il quale il sistema è stato sospeso (nel caso di sospensione in RAM o <i>ibernazione</i> su disco. Questo orologio è specifico di Linux, ed è disponibile a partire dal kernel 2.6.39.
CLOCK_REALTIME_ALARM	Identico a CLOCK_REALTIME, ma se usato per un timer il sistema sarà riattivato anche se è in sospensione. Questo orologio è specifico di Linux, ed è disponibile a partire dal kernel 3.0.
CLOCK_BOOTTIME_ALARM	Identico a CLOCK_BOOTTIME, ma se usato per un timer il sistema sarà riattivato anche se è in sospensione. Questo orologio è specifico di Linux, ed è disponibile a partire dal kernel 3.0.

Tabella 7.10: Valori possibili per una variabile di tipo `clockid_t` usata per indicare a quale tipo di orologio si vuole fare riferimento.

Le funzioni ritornano 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
EFAULT	l'indirizzo <code>tp</code> non è valido.
EINVAL	il valore specificato per <code>clockid</code> non è valido o il relativo orologio <i>real-time</i> non è supportato dal sistema.
EPERM	non si ha il permesso di impostare l'orologio indicato (solo per <code>clock_settime</code>).

Entrambe le funzioni richiedono che si specifichi come primo argomento il tipo di orologio su cui si vuole operare con uno dei valori di tab. 7.10 o con il risultato di una chiamata a `clock_getcpuclockid` (che tratteremo a breve), il secondo argomento invece è sempre il puntatore `tp` ad una struttura `timespec` (vedi fig. 4.16) che deve essere stata precedentemente allocata. Per `clock_settime` questa dovrà anche essere stata inizializzata con il valore che si vuole impostare sull'orologio, mentre per `clock_gettime` verrà restituito al suo interno il valore corrente dello stesso.

Si tenga presente inoltre che per eseguire un cambiamento sull'orologio generale di sistema `CLOCK_REALTIME` occorrono i privilegi amministrativi;¹⁶ inoltre ogni cambiamento ad esso apportato non avrà nessun effetto sulle temporizzazioni effettuate in forma relativa, come quelle impostate sulle quantità di *process time* o per un intervallo di tempo da trascorrere, ma solo su quelle che hanno richiesto una temporizzazione ad un istante preciso (in termini di *calendar time*). Si tenga inoltre presente che nel caso di Linux `CLOCK_REALTIME` è l'unico orologio per cui si può effettuare una modifica, infatti nonostante lo standard preveda la possibilità di

¹⁶ed in particolare la *capability* `CAP_SYS_TIME`.

modifiche anche per `CLOCK_PROCESS_CPUTIME_ID` e `CLOCK_THREAD_CPUTIME_ID`, il kernel non le consente.

Oltre alle due funzioni precedenti, lo standard POSIX prevede una terza funzione di sistema che consenta di ottenere la risoluzione effettiva fornita da un certo orologio, la funzione è `clock_getres` ed il suo prototipo è:

```
#include <time.h>
int clock_getres(clockid_t clockid, struct timespec *res)
                                                    Legge la risoluzione di un orologio real-time.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT l'indirizzo di `res` non è valido.

EINVAL il valore specificato per `clockid` non è valido.

La funzione richiede come primo argomento l'indicazione dell'orologio di cui si vuole conoscere la risoluzione (effettuata allo stesso modo delle due precedenti) e questa verrà restituita in una struttura `timespec` all'indirizzo puntato dall'argomento `res`.

Come accennato il valore di questa risoluzione dipende sia dall'hardware disponibile che dalla implementazione delle funzioni, e costituisce il limite minimo di un intervallo di tempo che si può indicare. Qualunque valore si voglia utilizzare nelle funzioni di impostazione che non corrisponda ad un multiplo intero di questa risoluzione, sarà troncato in maniera automatica.

Gli orologi elencati nella seconda sezione di tab. 7.10 sono delle estensioni specifiche di Linux, create per rispondere ad alcune esigenze specifiche, come quella di tener conto di eventuali periodi di sospensione del sistema, e presenti solo nelle versioni più recenti del kernel. In particolare gli ultimi due, contraddistinti dal suffisso `_ALARM`, hanno un impiego particolare, derivato dalle esigenze emerse con Android per l'uso di Linux sui cellulari, che consente di creare timer che possono scattare, riattivando il sistema, anche quando questo è in sospensione. Per il loro utilizzo è prevista la necessità di una capacità specifica, `CAP_WAKE_ALARM` (vedi sez. 9.1.1).

Si tenga presente inoltre che con l'introduzione degli *high resolution timer* i due orologi `CLOCK_PROCESS_CPUTIME_ID` e `CLOCK_THREAD_CPUTIME_ID` fanno riferimento ai contatori presenti in opportuni registri interni del processore; questo sui sistemi multiprocessore può avere delle ripercussioni sulla precisione delle misure di tempo che vanno al di là della risoluzione teorica ottenibile con `clock_getres`, che può essere ottenuta soltanto quando si è sicuri che un processo (o un *thread*) sia sempre stato eseguito sullo stesso processore.

Con i sistemi multiprocessore infatti ogni singola CPU ha i suoi registri interni, e se ciascuna di esse utilizza una base di tempo diversa (se cioè il segnale di temporizzazione inviato ai processori non ha una sola provenienza) in genere ciascuna di queste potrà avere delle frequenze leggermente diverse, e si otterranno pertanto dei valori dei contatori correlati fra loro, senza nessuna possibilità di sincronizzazione.

Il problema si presenta, in forma più lieve, anche se la base di tempo è la stessa, dato che un sistema multiprocessore non avvia mai tutte le CPU allo stesso istante, si potrà così avere di nuovo una differenza fra i contatori, soggetta però soltanto ad uno sfasamento costante. Per questo caso il kernel per alcune architetture ha del codice che consente di ridurre al minimo la differenza, ma non può essere comunque garantito che questa si annulli (anche se in genere risulta molto piccola e trascurabile nella gran parte dei casi).

Per poter gestire questo tipo di problematiche lo standard ha previsto una apposita funzione che sia in grado di ottenere l'identificativo dell'orologio associato al *process time* di un processo, la funzione è `clock_getcpuclockid` ed il suo prototipo è:

```
#include <time.h>
int clock_getcpuclockid(pid_t pid, clockid_t *clockid)
    Ottiene l'identificatore dell'orologio di CPU usato da un processo.
```

La funzione ritorna 0 in caso di successo ed un numero positivo per un errore, nel qual caso `errno` assumerà uno dei valori:

ENOSYS non c'è il supporto per ottenere l'orologio relativo al *process time* di un altro processo, e `pid` non corrisponde al processo corrente.

EPERM il chiamante non ha il permesso di accedere alle informazioni relative al processo `pid`, avviene solo se è disponibile il supporto per leggere l'orologio relativo ad un altro processo.

ESRCH non esiste il processo `pid`.

La funzione ritorna l'identificativo di un orologio di sistema associato ad un processo indicato tramite l'argomento `pid`. Un utente normale, posto che il kernel sia sufficientemente recente da supportare questa funzionalità, può accedere soltanto ai dati relativi ai propri processi.

Del tutto analoga a `clock_getcpuclockid`, ma da utilizzare per ottenere l'orologio associato ad un *thread* invece che a un processo, è `pthread_getcpuclockid`,¹⁷ il cui prototipo è:

```
#include <pthread.h>
#include <time.h>
int pthread_getcpuclockid(pthread_t thread, clockid_t *clockid)
    Ottiene l'identificatore dell'orologio di CPU associato ad un thread.
```

La funzione ritorna 0 in caso di successo ed un numero positivo per un errore, nel qual caso `errno` assumerà uno dei valori:

ENOENT la funzione non è supportata dal sistema.

ESRCH non esiste il *thread* identificato

Con l'introduzione degli orologi ad alta risoluzione è divenuto possibile ottenere anche una gestione più avanzata degli allarmi; abbiamo già visto in sez. 7.3.4 come l'interfaccia di `setitimer` derivata da BSD presenti delle serie limitazioni, come la possibilità di perdere un segnale sotto carico, tanto che nello standard POSIX.1-2008 questa viene marcata come obsoleta, e ne viene fortemente consigliata la sostituzione con nuova interfaccia definita dallo standard POSIX.1-2001 che va sotto il nome di *POSIX Timer API*. Questa interfaccia è stata introdotta a partire dal kernel 2.6, anche se il supporto di varie funzionalità da essa previste è stato aggiunto solo in un secondo tempo.

Una delle principali differenze della nuova interfaccia è che un processo può utilizzare un numero arbitrario di timer; questi vengono creati (ma non avviati) tramite la funzione di sistema `timer_create`, il cui prototipo è:

```
#include <signal.h>
#include <time.h>
int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid)
    Crea un nuovo timer POSIX.
```

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN fallimento nel tentativo di allocare le strutture dei timer.

EINVAL uno dei valori specificati per `clockid` o per i campi `sigev_notify`, `sigev_signo` o `sigev_notify_thread_id` di `evp` non è valido.

ENOMEM errore di allocazione della memoria.

¹⁷per poterla utilizzare, come per qualunque funzione che faccia riferimento ai *thread*, occorre effettuare il collegamento alla relativa libreria di gestione compilando il programma con `-lpthread`.

La funzione richiede tre argomenti: il primo argomento serve ad indicare quale tipo di orologio si vuole utilizzare e prende uno dei valori di tab. 7.10; di detti valori però non è previsto l'uso di `CLOCK_MONOTONIC_RAW` mentre `CLOCK_PROCESS_CPUTIME_ID` e `CLOCK_THREAD_CPUTIME_ID` sono disponibili solo a partire dal kernel 2.6.12. Si può così fare riferimento sia ad un tempo assoluto che al tempo utilizzato dal processo (o *thread*) stesso. Si possono inoltre utilizzare, posto di avere un kernel che li supporti, gli orologi aggiuntivi della seconda parte di tab. 7.10.

Il secondo argomento richiede una trattazione più dettagliata, in quanto introduce una struttura di uso generale, `sigevent`, che viene utilizzata anche da altre funzioni, come quelle per l'I/O asincrono (vedi sez. 10.3.3) o le code di messaggi POSIX (vedi sez. 11.4.2) e che serve ad indicare in maniera generica un meccanismo di notifica.

```

struct sigevent {
    int         sigev_notify;    /* Notification method */
    int         sigev_signo;    /* Timer expiration signal */
    union signal sigev_value;    /* Value accompanying signal or
                                passed to thread function */
    /* Function used for thread notifications (SIGEV_THREAD) */
    void        (*sigev_notify_function) (union signal);
    /* Attributes for notification thread (SIGEV_THREAD) */
    void        *sigev_notify_attributes;
    /* ID of thread to signal (SIGEV_THREAD_ID) */
    pid_t       sigev_notify_thread_id;
};

```

Figura 7.15: La struttura `sigevent`, usata per specificare in maniera generica diverse modalità di notifica degli eventi.

La struttura `sigevent` (accessibile includendo `time.h`) è riportata in fig. 7.15, la definizione effettiva dipende dall'implementazione, quella mostrata è la versione descritta nella pagina di manuale di `timer_create`. Il campo `sigev_notify` è il più importante essendo quello che indica le modalità della notifica, gli altri dipendono dal valore che si è specificato per `sigev_notify`, si sono riportati in tab. 7.11. La scelta del meccanismo di notifica viene fatta impostando uno dei valori di tab. 7.11 per `sigev_notify`, e fornendo gli eventuali ulteriori argomenti necessari a secondo della scelta effettuata. Diventa così possibile indicare l'uso di un segnale o l'esecuzione (nel caso di uso dei *thread*) di una funzione di modifica in un *thread* dedicato.

Nel caso di `timer_create` occorrerà passare alla funzione come secondo argomento l'indirizzo di una di queste strutture per indicare le modalità con cui si vuole essere notificati della scadenza del timer, se non si specifica nulla (passando un valore `NULL`) verrà inviato il segnale `SIGALRM` al processo corrente, o per essere più precisi verrà utilizzato un valore equivalente all'aver specificato `SIGEV_SIGNAL` per `sigev_notify`, `SIGALRM` per `sigev_signo` e l'identificatore del timer come valore per `sigev_value.sival_int`.

Il terzo argomento deve essere l'indirizzo di una variabile di tipo `timer_t` dove sarà scritto l'identificativo associato al timer appena creato, da usare in tutte le successive funzioni di gestione. Una volta creato questo identificativo resterà univoco all'interno del processo stesso fintanto che il timer non viene cancellato.

¹⁸nel caso dei *timer* questa funzionalità è considerata un esempio di pessima implementazione di una interfaccia, richiesta dallo standard POSIX, ma da evitare totalmente nell'uso ordinario, a causa della possibilità di creare disservizi generando una gran quantità di processi, tanto che ne è stata richiesta addirittura la rimozione.

Valore	Significato
SIGEV_NONE	Non viene inviata nessuna notifica.
SIGEV_SIGNAL	La notifica viene effettuata inviando al processo chiamante il segnale specificato dal campo <code>sigev_signo</code> ; se il gestore di questo segnale è stato installato con <code>SA_SIGINFO</code> gli verrà restituito il valore specificato con <code>sigev_value</code> (una <code>union sigval</code> , la cui definizione è in fig. 7.14) come valore del campo <code>si_value</code> di <code>siginfo_t</code> .
SIGEV_THREAD	La notifica viene effettuata creando un nuovo <i>thread</i> che esegue la funzione di notifica specificata da <code>sigev_notify_function</code> con argomento <code>sigev_value</code> . Se questo è diverso da <code>NULL</code> , il <i>thread</i> viene creato con gli attributi specificati da <code>sigev_notify_attribute</code> . ¹⁸
SIGEV_THREAD_ID	Invia la notifica come segnale (con le stesse modalità di <code>SIGEV_SIGNAL</code>) che però viene recapitato al <i>thread</i> indicato dal campo <code>sigev_notify_thread_id</code> . Questa modalità è una estensione specifica di Linux, creata come supporto per le librerie di gestione dei <i>thread</i> , pertanto non deve essere usata da codice normale.

Tabella 7.11: Valori possibili per il campo `sigev_notify` in una struttura `sigevent`.

Si tenga presente che eventuali POSIX timer creati da un processo non vengono ereditati dai processi figli creati con `fork` e che vengono cancellati nella esecuzione di un programma diverso attraverso una delle funzioni `exec`. Si tenga presente inoltre che il kernel prealloca l'uso di un segnale *real-time* per ciascun timer che viene creato con `timer_create`; dato che ciascuno di essi richiede un posto nella coda dei segnali *real-time*, il numero massimo di timer utilizzabili da un processo è limitato dalle dimensioni di detta coda, ed anche, qualora questo sia stato impostato, dal limite `RLIMIT_SIGPENDING`.

Una volta creato il timer `timer_create` ed ottenuto il relativo identificatore, si può attivare o disattivare un allarme (in gergo *armare* o *disarmare* il timer) con la funzione di sistema `timer_settime`, il cui prototipo è:

```
#include <signal.h>
#include <time.h>
int timer_settime(timer_t timerid, int flags, const struct itimerspec *new_value,
struct itimerspec *old_value)
```

Arma o disarma un timer POSIX.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT si è specificato un indirizzo non valido per `new_value` o `old_value`.

EINVAL all'interno di `new_value.value` si è specificato un tempo negativo o un numero di nanosecondi maggiore di 999999999.

La funzione richiede che si indichi la scadenza del timer con l'argomento `new_value`, che deve essere specificato come puntatore ad una struttura di tipo `itimerspec`, la cui definizione è riportata in fig. 7.16; se il puntatore `old_value` è diverso da `NULL` il valore corrente della scadenza verrà restituito in una analoga struttura, ovviamente in entrambi i casi le strutture devono essere state allocate.

```
struct itimerspec {
    struct timespec it_interval; /* Timer interval */
    struct timespec it_value;    /* Initial expiration */
};
```

Figura 7.16: La struttura `itimerspec`, usata per specificare la scadenza di un allarme.

Ciascuno dei due campi di `itimerspec` indica un tempo, da specificare con una precisione fino al nanosecondo tramite una struttura `timespec` (la cui definizione è riportata fig. 4.16). Il campo `it_value` indica la prima scadenza dell'allarme. Di default, quando il valore di `flags` è nullo, questo valore viene considerato come un intervallo relativo al tempo corrente, il primo allarme scatterà cioè dopo il numero di secondi e nanosecondi indicati da questo campo. Se invece si usa per `flags` il valore `TIMER_ABSTIME`, che al momento è l'unico valore valido per `flags`, allora `it_value` viene considerato come un valore assoluto rispetto al valore usato dall'orologio a cui è associato il timer.

Quindi a seconda dei casi si potrà impostare un timer o con un tempo assoluto, quando si opera rispetto all'orologio di sistema (nel qual caso il valore deve essere in secondi e nanosecondi dalla *epoch*) o con un numero di secondi o nanosecondi rispetto alla partenza di un orologio di CPU, quando si opera su uno di questi. Infine un valore nullo di `it_value`, dove per nullo si intende con valori nulli per entrambi i campi `tv_sec` e `tv_nsec`, può essere utilizzato, indipendentemente dal tipo di orologio utilizzato, per disarmare l'allarme.

Il campo `it_interval` di `itimerspec` viene invece utilizzato per impostare un allarme periodico. Se il suo valore è nullo, se cioè sono nulli tutti e due i due campi `tv_sec` e `tv_nsec` di detta struttura `timespec`, l'allarme scatterà una sola volta secondo quando indicato con `it_value`, altrimenti il valore specificato nella struttura verrà preso come l'estensione del periodo di ripetizione della generazione dell'allarme, che proseguirà indefinitamente fintanto che non si disarmi il timer.

Se il timer era già stato armato la funzione sovrascrive la precedente impostazione, se invece si indica come prima scadenza un tempo già passato, l'allarme verrà notificato immediatamente e al contempo verrà incrementato il contatore dei superamenti. Questo contatore serve a fornire una indicazione al programma che riceve l'allarme su un eventuale numero di scadenze che sono passate prima della ricezione della notifica dell'allarme.

È infatti possibile, qualunque sia il meccanismo di notifica scelto, che quest'ultima venga ricevuta dopo che il timer è scaduto più di una volta, specialmente se si imposta un timer con una ripetizione a frequenza elevata. Nel caso dell'uso di un segnale infatti il sistema mette in coda un solo segnale per timer,¹⁹ e se il sistema è sotto carico o se il segnale è bloccato, prima della sua ricezione può passare un intervallo di tempo sufficientemente lungo ad avere scadenze multiple, e lo stesso può accadere anche se si usa un *thread* di notifica.

Per questo motivo il gestore del segnale o il *thread* di notifica può ottenere una indicazione di quante volte il timer è scaduto dall'invio della notifica utilizzando la funzione di sistema `timer_getoverrun`, il cui prototipo è:

```
#include <time.h>
int timer_getoverrun(timer_t timerid)
```

Ottiene il numero di scadenze di un timer POSIX.

La funzione ritorna il numero di scadenze di un timer in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` `timerid` non indica un timer valido.

La funzione ritorna il numero delle scadenze avvenute, che può anche essere nullo se non ve ne sono state. Come estensione specifica di Linux,²⁰ quando si usa un segnale come meccanismo di notifica, si può ottenere direttamente questo valore nel campo `si_overrun` della struttura `siginfo_t` (illustrata in fig. 7.9) restituita al gestore del segnale installato con `sigaction`; in

¹⁹questo indipendentemente che si tratti di un segnale ordinario o *real-time*, per questi ultimi sarebbe anche possibile inviare un segnale per ogni scadenza, questo però non viene fatto per evitare il rischio, tutt'altro che remoto, di riempire la coda.

²⁰in realtà lo standard POSIX.1-2001 prevede gli *overrun* solo per i segnali e non ne parla affatto in riferimento ai *thread*.

questo modo non è più necessario eseguire successivamente una chiamata a questa funzione per ottenere il numero delle scadenze. Al gestore del segnale viene anche restituito, come ulteriore informazione, l'identificativo del timer, in questo caso nel campo `si_timerid`.

Qualora si voglia rileggere lo stato corrente di un timer, ed ottenere il tempo mancante ad una sua eventuale scadenza, si deve utilizzare la funzione di sistema `timer_gettime`, il cui prototipo è:

```
#include <time.h>
int timer_gettime(timer_t timerid, int flags, struct itimerspec *curr_value)
                                                    Legge lo stato di un timer POSIX.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT si è specificato un indirizzo non valido per `curr_value`.

EINVAL `timerid` non indica un timer valido.

La funzione restituisce nella struttura `itimerspec` puntata da `curr_value` il tempo restante alla prossima scadenza nel campo `it_value`. Questo tempo viene sempre indicato in forma relativa, anche nei casi in cui il timer era stato precedentemente impostato con `TIMER_ABSTIME` indicando un tempo assoluto. Il ritorno di un valore nullo nel campo `it_value` significa che il timer è disarmato o è definitivamente scaduto.

Nel campo `it_interval` di `curr_value` viene invece restituito, se questo era stato impostato, il periodo di ripetizione del timer. Anche in questo caso il ritorno di un valore nullo significa che il timer non era stato impostato per una ripetizione e doveva operare, come suol dirsi, a colpo singolo (in gergo *one shot*).

Infine, quando un timer non viene più utilizzato, lo si può cancellare, rimuovendolo dal sistema e recuperando le relative risorse, effettuando in sostanza l'operazione inversa rispetto a `timer_create`. Per questo compito lo standard prevede una apposita funzione di sistema, `timer_delete`, il cui prototipo è:

```
#include <time.h>
int timer_delete(timer_t timerid)
                                                    Cancella un timer POSIX.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL `timerid` non indica un timer valido.

La funzione elimina il timer identificato da `timerid`, disarmandolo se questo era stato attivato. Nel caso, poco probabile ma comunque possibile, che un timer venga cancellato prima della ricezione del segnale pendente per la notifica di una scadenza, il comportamento del sistema è indefinito.

Infine a partire dal kernel 2.6 e per le versioni della *libc* superiori alla 2.1, si può utilizzare la nuova interfaccia dei timer POSIX anche per le funzioni di attesa, per questo è disponibile la funzione di sistema `clock_nanosleep`, il cui prototipo è:

```
#include <time.h>
int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request,
                  struct timespec *remain)
                                                    Pone il processo in pausa per un tempo specificato.
```

La funzione ritorna 0 in caso di successo ed un valore positivo per un errore, espresso dai valori:

EINTR la funzione è stata interrotta da un segnale.

EINVAL si è specificato un numero di secondi negativo o un numero di nanosecondi maggiore di 999.999.999 o indicato un orologio non valido.

ed inoltre EFAULT nel suo significato generico.

I due argomenti `request` e `remain` sono identici agli analoghi di `nanosleep` che abbiamo visto in sez. 7.3.5, ed hanno lo stesso significato. L'argomento `clock_id` consente di indicare quale orologio si intende utilizzare per l'attesa con uno dei valori della prima parte di tab. 7.10 (eccetto `CLOCK_THREAD_CPUTIME_ID`). L'argomento `flags` consente di modificare il comportamento della funzione, il suo unico valore valido al momento è `TIMER_ABSTIME` che, come per `timer_settime` indica di considerare il tempo indicato in `request` come assoluto anziché relativo.

Il comportamento della funzione è analogo a `nanosleep`, se la chiamata viene interrotta il tempo rimanente viene restituito in `remain`. Utilizzata normalmente con attese relative può soffrire degli stessi problemi di deriva di cui si è parlato in sez. 7.3.5 dovuti ad interruzioni ripetute per via degli arrotondamenti fatti a questo tempo. Ma grazie alla possibilità di specificare tempi assoluti con `flags` si può ovviare a questo problema ricavando il tempo corrente con `clock_gettime`, aggiungendovi l'intervallo di attesa, ed impostando questa come tempo assoluto.

Si tenga presente che se si è usato il valore `TIMER_ABSTIME` per `flags` e si è indicato un tempo assoluto che è già passato la funzione ritorna immediatamente senza nessuna sospensione. In caso di interruzione da parte di un segnale il tempo rimanente viene restituito in `remain` soltanto se questo non è un puntatore `NULL` e non si è specificato `TIMER_ABSTIME` per `flags`.

7.5.3 Ulteriori funzioni di gestione

In questo ultimo paragrafo esamineremo le rimanenti funzioni di gestione dei segnali non descritte finora, relative agli aspetti meno utilizzati e più “esoterici” della interfaccia.

La prima di queste funzioni è la funzione di sistema `sigpending`, anch'essa introdotta dallo standard POSIX.1, il suo prototipo è:

```
#include <signal.h>
int sigpending(sigset_t *set)
```

Legge l'insieme dei segnali pendenti.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà solo il valore `EFAULT` nel suo significato generico.

La funzione permette di ricavare quali sono i segnali pendenti per il processo in corso, cioè i segnali che sono stati inviati dal kernel ma non sono stati ancora ricevuti dal processo in quanto bloccati. Non esiste una funzione equivalente nella vecchia interfaccia, ma essa è tutto sommato poco utile, dato che essa può solo assicurare che un segnale è stato inviato, dato che escluderne l'avvenuto invio al momento della chiamata non significa nulla rispetto a quanto potrebbe essere in un qualunque momento successivo.

Una delle caratteristiche di BSD, disponibile anche in Linux, è la possibilità di usare uno *stack* alternativo per i segnali; è cioè possibile fare usare al sistema un altro *stack* (invece di quello relativo al processo, vedi sez. 2.2.2) solo durante l'esecuzione di un gestore. L'uso di uno *stack* alternativo è del tutto trasparente ai gestori, occorre però seguire una certa procedura:

1. allocare un'area di memoria di dimensione sufficiente da usare come *stack* alternativo;
2. usare la funzione `sigaltstack` per rendere noto al sistema l'esistenza e la locazione dello *stack* alternativo;
3. quando si installa un gestore occorre usare `sigaction` specificando il flag `SA_ONSTACK` (vedi tab. 7.6) per dire al sistema di usare lo *stack* alternativo durante l'esecuzione del gestore.

In genere il primo passo viene effettuato allocando un'opportuna area di memoria con `malloc`; in `signal.h` sono definite due costanti, `SIGSTKSZ` e `MINSIGSTKSZ`, che possono essere utilizzate per allocare una quantità di spazio opportuna, in modo da evitare overflow. La prima delle due è la dimensione canonica per uno *stack* di segnali e di norma è sufficiente per tutti gli usi normali.

La seconda è lo spazio che occorre al sistema per essere in grado di lanciare il gestore e la dimensione di uno *stack* alternativo deve essere sempre maggiore di questo valore. Quando si conosce esattamente quanto è lo spazio necessario al gestore gli si può aggiungere questo valore per allocare uno *stack* di dimensione sufficiente.

Come accennato, per poter essere usato, lo *stack* per i segnali deve essere indicato al sistema attraverso la funzione `sigaltstack`; il suo prototipo è:

```
#include <signal.h>
int sigaltstack(const stack_t *ss, stack_t *oss)
    Installa uno stack alternativo per i gestori di segnali.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT	uno degli indirizzi degli argomenti non è valido.
EINVAL	<code>ss</code> non è nullo e <code>ss_flags</code> contiene un valore diverso da zero che non è <code>SS_DISABLE</code> .
ENOMEM	la dimensione specificata per il nuovo <i>stack</i> è minore di <code>MINSIGSTKSZ</code> .
EPERM	si è cercato di cambiare lo <i>stack</i> alternativo mentre questo è attivo (cioè il processo è in esecuzione su di esso).

La funzione prende come argomenti puntatori ad una struttura di tipo `stack_t`, definita in fig. 7.17. I due valori `ss` e `oss`, se non nulli, indicano rispettivamente il nuovo *stack* da installare e quello corrente (che viene restituito dalla funzione per un successivo ripristino).

```
typedef struct {
    void *ss_sp;      /* Base address of stack */
    int   ss_flags;    /* Flags */
    size_t ss_size;    /* Number of bytes in stack */
} stack_t;
```

Figura 7.17: La struttura `stack_t`.

Il campo `ss_sp` di `stack_t` indica l'indirizzo base dello *stack*, mentre `ss_size` ne indica la dimensione; il campo `ss_flags` invece indica lo stato dello *stack*. Nell'indicare un nuovo *stack* occorre inizializzare `ss_sp` e `ss_size` rispettivamente al puntatore e alla dimensione della memoria allocata, mentre `ss_flags` deve essere nullo. Se invece si vuole disabilitare uno *stack* occorre indicare `SS_DISABLE` come valore di `ss_flags` e gli altri valori saranno ignorati.

Se `oss` non è nullo verrà restituito dalla funzione indirizzo e dimensione dello *stack* corrente nei relativi campi, mentre `ss_flags` potrà assumere il valore `SS_ONSTACK` se il processo è in esecuzione sullo *stack* alternativo (nel qual caso non è possibile cambiarlo) e `SS_DISABLE` se questo non è abilitato.

In genere si installa uno *stack* alternativo per i segnali quando si teme di avere problemi di esaurimento dello *stack* standard o di superamento di un limite (vedi sez. 6.3.2) imposto con chiamate del tipo `setrlimit(RLIMIT_STACK, &rlim)`. In tal caso infatti si avrebbe un segnale di `SIGSEGV`, che potrebbe essere gestito soltanto avendo abilitato uno *stack* alternativo.

Si tenga presente che le funzioni chiamate durante l'esecuzione sullo *stack* alternativo continueranno ad usare quest'ultimo, che, al contrario di quanto avviene per lo *stack* ordinario

dei processi, non si accresce automaticamente (ed infatti eccederne le dimensioni può portare a conseguenze imprevedibili). Si ricordi infine che una chiamata ad una funzione della famiglia `exec` cancella ogni *stack* alternativo.

Abbiamo visto in fig. 7.6 come si possa usare `longjmp` per uscire da un gestore rientrando direttamente nel corpo del programma, sappiamo però che nell'esecuzione di un gestore il segnale che l'ha invocato viene bloccato, e abbiamo detto che possiamo ulteriormente modificarlo con `sigprocmask`.

Resta quindi il problema di cosa succede alla maschera dei segnali quando si esce da un gestore usando questa funzione. Il comportamento dipende dall'implementazione. In particolare la semantica usata da BSD prevede che sia ripristinata la maschera dei segnali precedente l'invocazione, come per un normale ritorno, mentre quella usata da System V no.

Lo standard POSIX.1 non specifica questo comportamento per `setjmp` e `longjmp`, ed il comportamento della *glibc* dipende da quale delle caratteristiche si sono abilitate con le macro viste in sez. 1.3.7.

Lo standard POSIX però prevede anche la presenza di altre due funzioni `sigsetjmp` e `siglongjmp`, che permettono di decidere in maniera esplicita quale dei due comportamenti il programma deve assumere; i loro prototipi sono:

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savesigs)
                                Salva il contesto dello stack e la maschera dei segnali.
void siglongjmp(sigjmp_buf env, int val)
                                Ripristina il contesto dello stack e la maschera dei segnali.
```

La funzioni sono identiche alle analoghe `setjmp` e `longjmp` di sez. 2.4.3 ed hanno gli stessi errori e valori di uscita.

Le due funzioni prendono come primo argomento la variabile su cui viene salvato il contesto dello *stack* per permettere il salto non-locale; nel caso specifico essa è di tipo `sigjmp_buf`, e non `jmp_buf` come per le analoghe di sez. 2.4.3 in quanto in questo caso viene salvata anche la maschera dei segnali.

Nel caso di `sigsetjmp`, se si specifica un valore di `savesigs` diverso da zero la maschera dei valori verrà salvata in `env` insieme al contesto dello *stack* usato per il salto non locale. Se così si è fatto la maschera dei segnali verrà ripristinata in una successiva chiamata a `siglongjmp`. Quest'ultima, a parte l'uso di un valore di `env` di tipo `sigjmp_buf`, è assolutamente identica a `longjmp`.

7.5.4 I *pidfd* e l'invio di segnali *race-free*

Capitolo 8

Terminali e sessioni di lavoro

A lungo l'unico modo per interagire con sistema di tipo Unix è stato tramite l'interfaccia dei terminali, ma anche oggi, nonostante la presenza di diverse interfacce grafiche, essi continuano ad essere estensivamente usati per il loro stretto legame la linea di comando.

Nella prima parte esamineremo i concetti base in cui si articola l'interfaccia dei terminali, a partire dal sistema del *job control* e delle sessioni di lavoro, toccando infine anche le problematiche dell'interazione con programmi non interattivi. Nella seconda parte tratteremo il funzionamento dell'I/O su terminale, e delle varie peculiarità che esso viene ad assumere nell'uso come interfaccia di accesso al sistema da parte degli utenti. La terza parte coprirà le tematiche relative alla creazione e gestione dei terminali virtuali, che consentono di replicare via software l'interfaccia dei terminali.

8.1 L'interazione con i terminali

I terminali sono l'interfaccia con cui fin dalla loro nascita i sistemi unix-like hanno gestito l'interazione con gli utenti, tramite quella riga di comando che li caratterizza da sempre. Ma essi hanno anche una rilevanza particolare perché quella dei terminali è l'unica interfaccia hardware usata dal kernel per comunicare direttamente con gli utenti, con la cosiddetta *console* di sistema, senza dover passare per un programma.

Originariamente si trattava di dispositivi specifici (i terminali seriali, se non addirittura le telescriventi). Oggi questa interfaccia viene in genere emulata o tramite programmi o con le cosiddette console virtuali associate a monitor e tastiera, ma esiste sempre la possibilità di associarla direttamente ad alcuni dispositivi, come eventuali linee seriali, ed in certi casi, come buona parte dei dispositivi embedded su cui gira Linux (come router, access point, ecc.) questa resta anche l'unica opzione per una *console* di sistema.

8.1.1 Il *job control*

Viene comunemente chiamato *job control* quell'insieme di funzionalità il cui scopo è quello di permettere ad un utente di poter sfruttare le capacità multitasking di un sistema Unix per eseguire in contemporanea più processi, pur potendo accedere, di solito, ad un solo terminale, avendo cioè un solo punto in cui si può avere accesso all'input ed all'output degli stessi. Con le interfacce grafiche di *X Window* e con i terminali virtuali via rete oggi tutto questo non è più vero, dato che si può accedere a molti terminali in contemporanea da una singola postazione di lavoro, ma il sistema è nato prima dell'esistenza di tutto ciò.

Il *job control* è una caratteristica opzionale, introdotta in BSD negli anni '80, e successivamente standardizzata da POSIX.1. La sua disponibilità nel sistema è verificabile attraverso il controllo della macro `_POSIX_JOB_CONTROL`. In generale il *job control* richiede il supporto sia da parte della shell (quasi tutte ormai lo hanno), che da parte del kernel. In particolare il kernel deve assicurare sia la presenza di un driver per i terminali abilitato al *job control* che quella dei relativi segnali illustrati in sez. 7.2.6.

In un sistema che supporta il *job control*, una volta completato il login, l'utente avrà a disposizione una shell dalla quale eseguire i comandi e potrà iniziare quella che viene chiamata una *sessione di lavoro*, che riunisce (vedi sez. 8.1.2) tutti i processi eseguiti all'interno dello stesso login (esamineremo tutto il processo in dettaglio in sez. 8.1.4).

Siccome la shell è collegata ad un solo terminale, che viene usualmente chiamato *terminale di controllo*, (vedi sez. 8.1.3) un solo comando alla volta, quello che viene detto in *foreground* o in *primo piano*, potrà scrivere e leggere dal terminale. La shell però può eseguire, aggiungendo una `&` alla fine del comando, più programmi in contemporanea, mandandoli come si dice, "in *background*" (letteralmente "sullo sfondo"), nel qual caso essi saranno eseguiti senza essere collegati al terminale.

Si noti come si sia parlato di comandi e non di programmi o processi. Fra le funzionalità della shell infatti c'è anche quella di consentire di concatenare più comandi in una sola riga con il *pipelining*, ed in tal caso verranno eseguiti più programmi. Inoltre, anche quando si invoca un singolo programma, questo potrà sempre lanciare eventuali sotto-processi per eseguire dei compiti specifici.

Per questo l'esecuzione di una riga di comando può originare più di un processo, quindi nella gestione del *job control* non si può far riferimento ai singoli processi. Per questo il kernel prevede la possibilità di raggruppare più processi in un cosiddetto *process group* (detto anche *raggruppamento di processi*, vedi sez. 8.1.2). Deve essere cura della shell far sì che tutti i processi che originano da una stessa riga di comando appartengano allo stesso raggruppamento di processi, in modo che le varie funzioni di controllo, ed i segnali inviati dal terminale, possano fare riferimento ad esso.

In generale all'interno di una sessione avremo un eventuale (può non esserci) *process group* in *foreground*, che riunisce i processi che possono accedere al terminale, e più *process group* in *background*, che non possono accedervi. Il *job control* prevede che quando un processo appartenente ad un raggruppamento in *background* cerca di accedere al terminale, venga inviato un segnale a tutti i processi del raggruppamento, in modo da bloccarli (vedi sez. 8.1.3).

Un comportamento analogo si ha anche per i segnali generati dai comandi di tastiera inviati dal terminale, che vengono inviati a tutti i processi del raggruppamento in *foreground*. In particolare C-z interrompe l'esecuzione del comando, che può poi essere mandato in *background* con il comando `bg`. Il comando `fg` consente invece di mettere in *foreground* un comando precedentemente lanciato in *background*. Si tenga presente che `bg` e `fg` sono comandi interni alla shell, che non comportano l'esecuzione di un programma esterno, ma operazioni di gestione compiute direttamente dalla shell stessa.

Di norma la shell si cura anche di notificare all'utente, di solito prima della stampa a video del prompt, lo stato dei vari processi. Essa infatti sarà in grado, grazie all'uso della funzione di sistema `waitpid` (vedi sez. 3.1.5), di rilevare sia i processi che sono terminati, sia i raggruppamenti che sono bloccati, in quest'ultimo caso si dovrà usare la specifica opzione `WUNTRACED`, secondo quanto già illustrato in sez. 3.1.5.

8.1.2 I *process group* e le sessioni

Come accennato in sez. 8.1.1 nel job control i processi vengono raggruppati in *process group* e *sessioni*; per far questo vengono utilizzati due ulteriori identificatori (oltre quelli visti in sez. 3.1.2) che il kernel associa a ciascun processo:¹ l'identificatore del *process group* e l'identificatore della *sessione*, che vengono indicati rispettivamente con le sigle *PGID* e *SID*, e sono mantenuti in variabili di tipo `pid_t`. I valori di questi identificatori possono essere visualizzati dal comando `ps` usando l'opzione `-j`.

Un *process group* è pertanto definito da tutti i processi che hanno lo stesso *PGID*; è possibile leggere il valore di questo identificatore con le funzioni di sistema `getpgid` e `getpgrp`, i cui prototipi sono:

<pre>#include <unistd.h> pid_t getpgid(pid_t pid)</pre>	Legge il <i>PGID</i> di un processo.
<pre>pid_t getpgrp(void)</pre>	Legge il <i>PGID</i> del processo corrente.
<p>Le funzioni ritornano il <i>PGID</i> richiesto in caso di successo, <code>getpgrp</code> ha sempre successo mentre <code>getpgid</code> restituisce <code>-1</code> per un errore, nel qual caso <code>errno</code> potrà assumere solo il valore <code>ESRCH</code> se il processo indicato non esiste.</p>	

Le due funzioni sono definite nello standard POSIX.1-2001, ma la prima deriva da SVr4 e la seconda da BSD4.2 dove però è previsto possa prendere un argomento per indicare il *PID* di un altro processo. Si può riottenere questo comportamento se si definisce la macro `_BSD_SOURCE` e non sono definite le altre macro che richiedono la conformità a POSIX, X/Open o SysV (vedi sez. 1.3).

La funzione `getpgid` permette di specificare il *PID* del processo di cui si vuole sapere il *PGID*. Un valore nullo per `pid` restituisce il *PGID* del processo corrente, che è il comportamento ordinario di `getpgrp`, che di norma equivalente a `getpgid(0)`.

In maniera analoga l'identificatore della sessione di un processo (il *SID*) può essere letto dalla funzione di sistema `getsid`, il cui prototipo è:

<pre>#include <unistd.h> pid_t getsid(pid_t pid)</pre>	Legge il <i>SID</i> di un processo.
<p>La funzione ritorna l'identificatore (un numero positivo) in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>EPERM</code> il processo selezionato non fa parte della stessa sessione del processo corrente (solo in alcune implementazioni).</p> <p><code>ESRCH</code> il processo selezionato non esiste.</p>	

La funzione è stata introdotta in Linux a partire dal kernel 1.3.44, il supporto nelle librerie del C è iniziato dalla versione 5.2.19. La funzione non era prevista originariamente da POSIX.1, che parla solo di processi leader di sessione, e non di identificatori di sessione, ma è prevista da SVr4 e fa parte di POSIX.1-2001. Per poterla utilizzare occorre definire la macro `_XOPEN_SOURCE` ad un valore maggiore o uguale di 500. Su Linux l'errore `EPERM` non viene mai restituito.

Entrambi gli identificatori, *SID* e *PGID*, vengono inizializzati nella creazione di ciascun processo con lo stesso valore che hanno nel processo padre, per cui un processo appena creato appartiene sempre allo stesso raggruppamento e alla stessa sessione del padre. Vedremo a

¹in Linux questi identificatori sono mantenuti nei campi `pgrp` e `session` della struttura `task_struct` definita in `include/linux/sched.h`.

breve come sia possibile creare più *process group* all'interno della stessa sessione, e spostare i processi dall'uno all'altro, ma sempre all'interno di una stessa sessione.

Ciascun raggruppamento di processi ha sempre un processo principale, il cosiddetto *process group leader* o più brevemente *group leader*, che è identificato dall'avere un *PGID* uguale al suo *PID*. In genere questo è il primo processo del raggruppamento, che si incarica di lanciare tutti gli altri. Un nuovo raggruppamento si crea con la funzione di sistema `setpgrp`, il cui prototipo è:

```
#include <unistd.h>
int setpgrp(void)
```

Rende un processo *group leader* di un nuovo gruppo.

La funzione ritorna il valore del nuovo *process group* e non sono previsti errori.

La funzione assegna al *PGID* il valore del *PID* del processo corrente, rendendolo in tal modo *group leader* di un nuovo raggruppamento. Tutti i successivi processi da esso creati apparterranno (a meno di non cambiare di nuovo il *PGID*) al nuovo raggruppamento.

La versione illustrata è quella usata nella definizione di POSIX.1, in BSD viene usata una funzione con questo nome, che però è identica a `setpgid`, che vedremo a breve, negli argomenti e negli effetti. Nella *glibc* viene sempre usata sempre questa definizione, a meno di non richiedere esplicitamente la compatibilità all'indietro con BSD, definendo la macro `_BSD_SOURCE` ed evitando di definire le macro che richiedono gli altri standard, come per `getpgrp`.

È inoltre possibile spostare un processo da un raggruppamento di processi ad un altro cambiandone il *PGID* con la funzione di sistema `setpgid`, il cui prototipo è:

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid)
```

Modifica il *PGID* di un processo.

La funzione ritorna il valore del nuovo *process group* in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES il processo di cui si vuole cambiare il *PGID* ha già eseguito una *exec*.
 EINVAL il valore di *pgid* è negativo.
 EPERM il cambiamento non è consentito.
 ESRCH il processo selezionato non esiste.

La funzione permette di cambiare il *PGID* del processo indicato dall'argomento `pid`, ma il cambiamento può essere effettuato solo se l'argomento `pgid` indica un *process group* che è nella stessa sessione del processo chiamante. Inoltre la funzione può essere usata soltanto sul processo corrente o su uno dei suoi figli, ed in quest'ultimo caso ha successo soltanto se questo non ha ancora eseguito una *exec*.² Specificando un valore nullo per `pid` si indica il processo corrente, mentre specificando un valore nullo per `pgid` si imposta il *process group* al valore del *PID* del processo selezionato, questo significa che `setpgrp` è equivalente a `setpgid(0, 0)`.

Di norma questa funzione viene usata dalla shell quando si usano delle pipeline, per mettere nello stesso *process group* tutti i programmi lanciati su ogni linea di comando; essa viene chiamata dopo una *fork* sia dal processo padre, per impostare il valore nel figlio, che da quest'ultimo, per sé stesso, in modo che il cambiamento di *process group* sia immediato per entrambi; una delle due chiamate sarà ridondante, ma non potendo determinare quale dei due processi viene eseguito per primo, occorre eseguirle comunque entrambe per evitare di esporsi ad una *race condition*.

²questa caratteristica è implementata dal kernel che mantiene allo scopo un altro campo, `did_exec`, nella struttura `task_struct`.

Si noti come nessuna delle funzioni esaminate finora permetta di spostare un processo da una sessione ad un'altra; infatti l'unico modo di far cambiare sessione ad un processo è quello di crearne una nuova con l'uso della funzione di sistema `setsid`, il cui prototipo è:

```
#include <unistd.h>
pid_t setsid(void)
```

Crea una nuova sessione sul processo corrente.

La funzione ritorna il valore del nuovo *SID* in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EPERM` il *PGID* e *PID* del processo coincidono.

La funzione imposta il *PGID* ed il *SID* del processo corrente al valore del suo *PID*, creando così una nuova sessione ed un nuovo *process group* di cui esso diventa leader (come per i *process group* un processo si dice *leader di sessione* se il suo *SID* è uguale al suo *PID*) ed unico componente.³ Inoltre la funzione distacca il processo da ogni terminale di controllo (torneremo sull'argomento in sez. 8.1.3) cui fosse in precedenza associato.

La funzione ha successo soltanto se il processo non è già leader di un *process group*, per cui per usarla di norma si esegue una `fork` e si esce, per poi chiamare `setsid` nel processo figlio, in modo che, avendo questo lo stesso *PGID* del padre ma un *PID* diverso, non ci siano possibilità di errore.

Potrebbe sorgere il dubbio che, per il riutilizzo dei valori dei *PID* fatto nella creazione dei nuovi processi (vedi sez. 3.1.2), il figlio venga ad assumere un valore corrispondente ad un *process group* esistente; questo viene evitato dal kernel che considera come disponibili per un nuovo *PID* solo valori che non corrispondono ad altri *PID*, *PGID* o *SID* in uso nel sistema. Questa funzione viene usata di solito nel processo di login (per i dettagli vedi sez. 8.1.4) per raggruppare in una sessione tutti i comandi eseguiti da un utente dalla sua shell.

8.1.3 Il terminale di controllo e il controllo di sessione

Come accennato in sez. 8.1.1, nel sistema del *job control* i processi all'interno di una sessione fanno riferimento ad un terminale di controllo (ad esempio quello su cui si è effettuato il login), sul quale effettuano le operazioni di lettura e scrittura, e dal quale ricevono gli eventuali segnali da tastiera. Nel caso di login grafico la cosa può essere più complessa, e di norma l'I/O è effettuato tramite il server X, ma ad esempio per i programmi, anche grafici, lanciati da un qualunque emulatore di terminale, sarà quest'ultimo a fare da terminale (virtuale) di controllo.

Per realizzare questa funzionalità lo standard POSIX.1 prevede che ad ogni sessione possa essere associato un terminale di controllo; in Linux questo viene realizzato mantenendo fra gli attributi di ciascun processo anche qual'è il suo terminale di controllo.⁴ In generale ogni processo eredita dal padre, insieme al *PGID* e al *SID* anche il terminale di controllo (vedi sez. 3.1.3). In questo modo tutti processi originati dallo stesso leader di sessione mantengono lo stesso terminale di controllo.

Alla creazione di una nuova sessione con `setsid` ogni associazione con il precedente terminale di controllo viene cancellata, ed il processo che è divenuto un nuovo leader di sessione dovrà essere associato ad un nuovo terminale di controllo.⁵ Questo viene fatto dal kernel la prima volta che il processo apre un terminale (cioè uno dei vari file di dispositivo `/dev/tty*`). In tal caso questo diventa automaticamente il terminale di controllo, ed il processo diventa il

³in Linux la proprietà è mantenuta in maniera indipendente con un apposito campo `leader` in `task_struct`.

⁴lo standard POSIX.1 non specifica nulla riguardo l'implementazione; in Linux anch'esso viene mantenuto nella solita struttura `task_struct`, nel campo `tty`.

⁵questo però solo quando necessario, cosa che, come vedremo in sez. 8.1.5, non è sempre vera.

processo di controllo di quella sessione. Questo avviene automaticamente a meno di non avere richiesto esplicitamente il contrario aprendo il terminale di controllo con il flag `O_NOCTTY` (vedi sez. 5.1.2). In questo Linux segue la semantica di SVr4; BSD invece richiede che il terminale venga allocato esplicitamente con una `ioctl` con il comando `TIOCSCTTY`.

In genere, a meno di redirezioni, nelle sessioni di lavoro il terminale di controllo è associato ai file standard (input, output ed error) dei processi nella sessione, ma solo quelli che fanno parte del cosiddetto raggruppamento di *foreground*, possono leggere e scrivere in certo istante. Per impostare il raggruppamento di *foreground* di un terminale si usa la funzione `tcsetpgrp`, il cui prototipo è:

<pre>#include <unistd.h> #include <termios.h> int tcsetpgrp(int fd, pid_t pgrp);</pre>	Imposta il <i>process group</i> di <i>foreground</i> .
<p>La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>ENOSYS</code> il sistema non supporta il <i>job control</i>.</p> <p><code>ENOTTY</code> il file <code>fd</code> non corrisponde al terminale di controllo del processo chiamante.</p> <p><code>EPERM</code> il <i>process group</i> specificato non è nella stessa sessione del processo chiamante.</p> <p>ed inoltre <code>EBADF</code> ed <code>EINVAL</code> nel loro significato generico.</p>	

La funzione imposta a `pgrp` il *process group* di *foreground* del terminale associato al file descriptor `fd`. La funzione può essere eseguita con successo solo da un processo che ha `fd` come terminale di controllo della propria sessione; inoltre `pgrp` deve essere un *process group* (non vuoto) appartenente alla stessa sessione del processo chiamante.

Come accennato in sez. 8.1.1, tutti i processi (e relativi raggruppamenti) che non fanno parte del gruppo di *foreground* sono detti in *background*; se uno si essi cerca di accedere al terminale di controllo provocherà l'invio da parte del kernel di uno dei due segnali `SIGTTIN` o `SIGTTOU` (a seconda che l'accesso sia stato in lettura o scrittura) a tutto il suo *process group*; dato che il comportamento di default di questi segnali (si riveda quanto esposto in sez. 7.2.6) è di fermare il processo, di norma questo comporta che tutti i membri del gruppo verranno fermati, ma non si avranno condizioni di errore.⁶ Se però si bloccano o ignorano i due segnali citati, le funzioni di lettura e scrittura falliranno con un errore di `EIO`.

In genere la funzione viene chiamata da un processo che è gruppo di *foreground* per passare l'accesso al terminale ad altri processi, quando viene chiamata da un processo che non è nel gruppo di *foreground*, a meno che questi non stia bloccando o ignorando il segnale `SIGTTOU`, detto segnale verrà inviato a tutti i processi del gruppo di cui il chiamante fa parte.

Un processo può controllare qual è il gruppo di *foreground* associato ad un terminale con la funzione `tcgetpgrp`, il cui prototipo è:

<pre>#include <unistd.h> #include <termios.h> pid_t tcgetpgrp(int fd);</pre>	Legge il <i>process group</i> di <i>foreground</i> .
<p>La funzione ritorna il <code>PGID</code> del gruppo di <i>foreground</i> in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p><code>ENOTTY</code> non c'è un terminale di controllo o <code>fd</code> non corrisponde al terminale di controllo del processo chiamante.</p> <p>ed inoltre <code>EBADF</code> nel suo significato generico.</p>	

⁶la shell in genere notifica comunque un avvertimento, avvertendo la presenza di processi bloccati grazie all'uso di `waitpid`.

La funzione legge il *process group* di *foreground* associato al file descriptor *fd*, che deve essere un terminale, restituendolo come risultato. Sia questa funzione che la precedente sono state introdotte con lo standard POSIX.1-2001, ma su Linux sono realizzate utilizzando le operazioni di *ioctl* con i comandi *TIOCGPRG* e *TIOCSPGRP*.

Si noti come entrambe le funzioni usino come argomento il valore di un file descriptor, il risultato comunque non dipende dal file descriptor che si usa ma solo dal terminale cui fa riferimento. Il kernel inoltre permette a ciascun processo di accedere direttamente al suo terminale di controllo attraverso il file speciale */dev/tty*, che per ogni processo è un sinonimo per il proprio terminale di controllo. Questo consente anche a processi che possono aver rediretto l'output di accedere al terminale di controllo, pur non disponendo più del file descriptor originario; un caso tipico è il programma *crypt* che accetta la redirectione sullo *standard input* di un file da decifrare, ma deve poi leggere la password dal terminale.

Un'altra caratteristica del terminale di controllo usata nel *job control* è che utilizzando su di esso le combinazioni di tasti speciali (C-z, C-c, C-y e C-|) si farà sì che il kernel invii i corrispondenti segnali (rispettivamente *SIGTSTP*, *SIGINT*, *SIGQUIT* e *SIGTERM*, trattati in sez. 7.2.6) a tutti i processi del raggruppamento di *foreground*; in questo modo la shell può gestire il blocco e l'interruzione dei vari comandi.

Per completare la trattazione delle caratteristiche del *job control* legate al terminale di controllo, occorre prendere in considerazione i vari casi legati alla terminazione anomala dei processi, che sono di norma gestite attraverso il segnale *SIGHUP*. Il nome del segnale deriva da *hungup*, termine che viene usato per indicare la condizione in cui il terminale diventa inutilizzabile, (letteralmente sarebbe *impiccagione*).

Quando si verifica questa condizione, ad esempio se si interrompe la linea, o va giù la rete o più semplicemente si chiude forzatamente la finestra di terminale su cui si stava lavorando, il kernel provvederà ad inviare il segnale di *SIGHUP* al processo di controllo. L'azione preimpostata in questo caso è la terminazione del processo, il problema che si pone è cosa accade agli altri processi nella sessione, che non hanno più un processo di controllo che possa gestire l'accesso al terminale, che potrebbe essere riutilizzato per qualche altra sessione.

Lo standard POSIX.1 prevede che quando il processo di controllo termina, che ciò avvenga o meno per un *hungup* del terminale (ad esempio si potrebbe terminare direttamente la shell con *kill*) venga inviato un segnale di *SIGHUP* ai processi del raggruppamento di *foreground*. In questo modo essi potranno essere avvisati che non esiste più un processo in grado di gestire il terminale (di norma tutto ciò comporta la terminazione anche di questi ultimi).

Restano però gli eventuali processi in *background*, che non ricevono il segnale; in effetti se il terminale non dovesse più servire essi potrebbero proseguire fino al completamento della loro esecuzione; ma si pone il problema di come gestire quelli che sono bloccati, o che si bloccano nell'accesso al terminale, in assenza di un processo che sia in grado di effettuare il controllo dello stesso.

Questa è la situazione in cui si ha quello che viene chiamato un *orphaned process group*. Lo standard POSIX.1 lo definisce come un *process group* i cui processi hanno come padri esclusivamente o altri processi nel raggruppamento, o processi fuori della sessione. Lo standard prevede inoltre che se la terminazione di un processo fa sì che un raggruppamento di processi diventi orfano e se i suoi membri sono bloccati, ad essi vengano inviati in sequenza i segnali di *SIGHUP* e *SIGCONT*.

La definizione può sembrare complicata, e a prima vista non è chiaro cosa tutto ciò abbia a che fare con il problema della terminazione del processo di controllo. Consideriamo allora cosa avviene di norma nel *job control*: una sessione viene creata con *setsid* che crea anche un nuovo *process group*. Per definizione quest'ultimo è sempre *orfano*, dato che il padre del leader di sessione è fuori dalla stessa e il nuovo *process group* contiene solo il leader di sessione.

Questo è un caso limite, e non viene emesso nessun segnale perché quanto previsto dallo standard riguarda solo i raggruppamenti che diventano orfani in seguito alla terminazione di un processo.⁷

Il leader di sessione provvederà a creare nuovi raggruppamenti che a questo punto non sono orfani in quanto esso resta padre per almeno uno dei processi del gruppo (gli altri possono derivare dal primo). Alla terminazione del leader di sessione però avremo che, come visto in sez. 3.1.4, tutti i suoi figli vengono adottati da `init`, che è fuori dalla sessione. Questo renderà orfani tutti i *process group* creati direttamente dal leader di sessione a meno di non aver spostato con `setpgid` un processo da un gruppo ad un altro, (cosa che di norma non viene fatta) i quali riceveranno, nel caso siano bloccati, i due segnali; `SIGCONT` ne farà proseguire l'esecuzione, ed essendo stato nel frattempo inviato anche `SIGHUP`, se non c'è un gestore per quest'ultimo, i processi bloccati verranno automaticamente terminati.

8.1.4 Dal login alla shell

L'organizzazione del sistema del job control è strettamente connessa alle modalità con cui un utente accede al sistema per dare comandi, collegandosi ad esso con un terminale, che sia questo realmente tale, come un VT100 collegato ad una seriale o virtuale, come quelli associati a schermo e tastiera o ad una connessione di rete. Dato che i concetti base sono gli stessi, e dato che alla fine le differenze sono nel dispositivo cui il kernel associa i file standard (vedi tab. 5.1) per l'I/O, tratteremo solo il caso classico del terminale, in generale nel caso di login via rete o di terminali lanciati dall'interfaccia grafica cambia anche il processo da cui ha origine l'esecuzione della shell.

Abbiamo già brevemente illustrato in sez. 1.1.2 le modalità con cui il sistema si avvia, e di come, a partire da `init`, vengano lanciati tutti gli altri processi. Adesso vedremo in maniera più dettagliata le modalità con cui il sistema arriva a fornire ad un utente la shell che gli permette di lanciare i suoi comandi su un terminale.

Nella maggior parte delle distribuzioni di GNU/Linux viene usata la procedura di avvio di System V;⁸ questa prevede che `init` legga dal file di configurazione `/etc/inittab` quali programmi devono essere lanciati, ed in quali modalità, a seconda del cosiddetto *run level*, anch'esso definito nello stesso file.

Tralasciando la descrizione del sistema dei *run level*, (per il quale si rimanda alla lettura delle pagine di manuale di `init` e di `inittab` o alla trattazione in sez. 5.3 di [?]) quello che comunque viene sempre fatto è di eseguire almeno una istanza di un programma che permetta l'accesso ad un terminale. Uno schema di massima della procedura è riportato in fig. 8.1.

Figura 8.1: Schema della procedura di login su un terminale.

Un terminale, che esso sia un terminale effettivo, attaccato ad una seriale o ad un altro tipo di porta di comunicazione, o una delle console virtuali associate allo schermo, o un terminale virtuale ad uso degli emulatori o delle sessioni di rete, viene sempre visto attraverso un apposito file di dispositivo che presenta una serie di caratteristiche comuni che vanno a costituire l'interfaccia generica di accesso ai terminali.

⁷l'emissione dei segnali infatti avviene solo nella fase di uscita del processo, come una delle operazioni legate all'esecuzione di `_exit`, secondo quanto illustrato in sez. 3.1.4.

⁸in realtà negli ultimi tempi questa situazione sta cambiando, e sono state proposte diversi possibili rimpiazzi per il tradizionale `init` di System V, come `upstart` o `systemd`, ma per quanto trattato in questa sezione il risultato finale non cambia, si avrà comunque il lancio di un programma che consenta l'accesso al terminale.

Per controllare un terminale fisico come la seriale o le console virtuali dello schermo si usa di solito il programma `getty` (o una delle sue varianti). Alla radice della catena che porta ad una shell per i comandi perciò c'è sempre `init` che esegue prima una `fork` e poi una `exec` per lanciare una istanza di questo programma, il tutto ripetuto per ciascuno dei terminali che si vogliono attivare, secondo quanto indicato dall'amministratore nel file di configurazione del programma, `/etc/inittab`.

Quando viene lanciato da `init` il programma parte con i privilegi di amministratore e con un ambiente vuoto; `getty` si cura di chiamare `setsid` per creare una nuova sessione ed un nuovo *process group*, e di aprire il terminale (che così diventa il terminale di controllo della sessione) in lettura sullo *standard input* ed in scrittura sullo *standard output* e sullo *standard error*; inoltre effettuerà, qualora servano, ulteriori impostazioni.⁹ Alla fine il programma stamperà un messaggio di benvenuto per poi porsi in attesa dell'immissione del nome di un utente.

Una volta che si sia immesso il nome di login `getty` esegue direttamente il programma `login` con una `execle`, passando come argomento la stringa con il nome, ed un ambiente opportunamente costruito che contenga quanto necessario; ad esempio di solito viene opportunamente inizializzata la variabile di ambiente `TERM` per identificare il terminale su cui si sta operando, a beneficio dei programmi che verranno lanciati in seguito.

A sua volta `login`, che mantiene i privilegi di amministratore, usa il nome dell'utente per effettuare una ricerca nel database degli utenti (in genere viene chiamata `getpwnam`, che abbiamo visto in sez. 6.2.1, per leggere la password e gli altri dati dal database degli utenti) e richiede una password. Se l'utente non esiste o se la password non corrisponde¹⁰ la richiesta viene ripetuta un certo numero di volte dopo di che `login` esce ed `init` provvede a rilanciare un'altra istanza di `getty`.

Se invece la password corrisponde `login` esegue `chdir` per impostare come directory di lavoro la *home directory* dell'utente, cambia i diritti di accesso al terminale (con `chown` e `chmod`) per assegnarne la titolarità all'utente ed al suo gruppo principale, assegnandogli al contempo i diritti di lettura e scrittura.¹¹ Inoltre il programma provvede a costruire gli opportuni valori per le variabili di ambiente, come `HOME`, `SHELL`, ecc. Infine attraverso l'uso di `setuid`, `setgid` e `initgroups` verrà cambiata l'identità del proprietario del processo, infatti, come spiegato in sez. 3.2.2, avendo invocato tali funzioni con i privilegi di amministratore, tutti gli `UID` ed i `GID` (reali, effettivi e salvati) saranno impostati a quelli dell'utente.

A questo punto `login` provvederà (fatte salve eventuali altre azioni iniziali, come la stampa di messaggi di benvenuto o il controllo della posta) ad eseguire con un'altra `exec` la shell, che si troverà con un ambiente già pronto con i file standard di tab. 5.1 impostati sul terminale, e pronta, nel ruolo di leader di sessione e di processo di controllo per il terminale, a gestire l'esecuzione dei comandi come illustrato in sez. 8.1.1.

Dato che il processo padre resta sempre `init` quest'ultimo potrà provvedere, ricevendo un `SIGCHLD` all'uscita della shell quando la sessione di lavoro è terminata, a rilanciare `getty` sul terminale per ripetere da capo tutto il procedimento.

⁹ad esempio, come qualcuno si sarà accorto scrivendo un nome di login in maiuscolo, può effettuare la conversione automatica dell'input in minuscolo, ponendosi in una modalità speciale che non distingue fra i due tipi di caratteri (a beneficio di alcuni vecchi terminali che non supportavano le minuscole).

¹⁰il confronto non viene effettuato con un valore in chiaro; quanto immesso da terminale viene invece a sua volta criptato, ed è il risultato che viene confrontato con il valore che viene mantenuto nel database degli utenti.

¹¹oggi queste operazioni, insieme ad altre relative alla contabilità ed alla tracciatura degli accessi, vengono gestite dalle distribuzioni più recenti in una maniera generica appoggiandosi a servizi di sistema come *ConsoleKit*, ma il concetto generale resta sostanzialmente lo stesso.

8.1.5 Interazione senza terminale: i *demoni* ed il *syslog*

Come sottolineato fin da sez. 1.1.1, in un sistema unix-like tutte le operazioni sono eseguite tramite processi, comprese quelle operazioni di sistema (come l'esecuzione dei comandi periodici, o la consegna della posta, ed in generale tutti i programmi di servizio) che non hanno niente a che fare con la gestione diretta dei comandi dell'utente.

Questi programmi, che devono essere eseguiti in modalità non interattiva e senza nessun intervento dell'utente, sono normalmente chiamati *demoni*, (o *daemons*), nome ispirato dagli omonimi spiritelli della mitologia greca che svolgevano compiti che gli dei trovavano noiosi, di cui parla anche Socrate (che sosteneva di averne uno al suo servizio).

Se però si lancia un programma demone dalla riga di comando in un sistema che supporta, come Linux, il *job control* esso verrà comunque associato ad un terminale di controllo e mantenuto all'interno di una sessione, e anche se può essere mandato in background e non eseguire più nessun I/O su terminale, si avranno comunque tutte le conseguenze che abbiamo trattato in sez. 8.1.3, in particolare l'invio dei segnali in corrispondenza dell'uscita del leader di sessione.

Per questo motivo un programma che deve funzionare come demone deve sempre prendere autonomamente i provvedimenti opportuni (come distaccarsi dal terminale e dalla sessione) ad impedire eventuali interferenze da parte del sistema del *job control*; questi sono riassunti in una lista di prescrizioni¹² da seguire quando si scrive un demone.

Pertanto, quando si lancia un programma che deve essere eseguito come demone occorrerà predisporlo in modo che esso compia le seguenti azioni:

1. Eseguire una *fork* e terminare immediatamente il processo padre proseguendo l'esecuzione nel figlio. In questo modo si ha la certezza che il figlio non è un *process group leader*, (avrà il *PGID* del padre, ma un *PID* diverso) e si può chiamare *setsid* con successo. Inoltre la shell considererà terminato il comando all'uscita del padre.
2. Eseguire *setsid* per creare una nuova sessione ed un nuovo raggruppamento di cui il processo diventa automaticamente il leader, che però non ha associato nessun terminale di controllo.
3. Assicurarsi che al processo non venga associato in seguito nessun nuovo terminale di controllo; questo può essere fatto sia avendo cura di usare sempre l'opzione *O_NOCTTY* nell'aprire i file di terminale, che eseguendo una ulteriore *fork* uscendo nel padre e proseguendo nel figlio. In questo caso, non essendo più quest'ultimo un leader di sessione non potrà ottenere automaticamente un terminale di controllo.
4. Eseguire una *chdir* per impostare la directory di lavoro del processo (su */* o su una directory che contenga dei file necessari per il programma), per evitare che la directory da cui si è lanciato il processo resti in uso e non sia possibile rimuoverla o smontare il filesystem che la contiene.
5. Impostare la maschera dei permessi (di solito con *umask(0)*) in modo da non essere dipendenti dal valore ereditato da chi ha lanciato originariamente il processo.
6. Chiudere tutti i file aperti che non servono più (in generale tutti); in particolare vanno chiusi i file standard che di norma sono ancora associati al terminale (un'altra opzione è quella di redirigerli verso */dev/null*).

In Linux buona parte di queste azioni possono venire eseguite invocando la funzione *daemon* (fornita dalla *glibc*), introdotta per la prima volta in BSD4.4; il suo prototipo è:

¹²ad esempio sia Stevens in [?], che la *Unix Programming FAQ* [?] ne riportano di sostanzialmente identiche.


```
#include <unistd.h>
int daemon(int nochdir, int noclose)
```

Rende il processo un demone.

La funzione ritorna (nel nuovo processo) 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori i valori impostati dalle sottostanti `fork` e `setsid`.

La funzione esegue una `fork`, per uscire subito, con `_exit`, nel padre, mentre l'esecuzione prosegue nel figlio che esegue subito una `setsid`. In questo modo si compiono automaticamente i passi 1 e 2 della precedente lista. Se `nochdir` è nullo la funzione imposta anche la directory di lavoro su `/`, se `noclose` è nullo i file standard vengono rediretti su `/dev/null` (corrispondenti ai passi 4 e 6); in caso di valori non nulli non viene eseguita nessuna altra azione.

Dato che un programma demone non può più accedere al terminale, si pone il problema di come fare per la notifica di eventuali errori, non potendosi più utilizzare lo *standard error*. Per il normale I/O infatti ciascun demone avrà le sue modalità di interazione col sistema e gli utenti a seconda dei compiti e delle funzionalità che sono previste; ma gli errori devono normalmente essere notificati all'amministratore del sistema.

Una soluzione può essere quella di scrivere gli eventuali messaggi su uno specifico file (cosa che a volte viene fatta comunque) ma questo comporta il grande svantaggio che l'amministratore dovrà tenere sotto controllo un file diverso per ciascun demone, e che possono anche generarsi conflitti di nomi. Per questo in BSD4.2 venne introdotto un servizio di sistema, il *syslog*, che oggi si trova su tutti i sistemi Unix, e che permette ai demoni di inviare messaggi all'amministratore in una maniera standardizzata.

Il servizio prevede vari meccanismi di notifica, e, come ogni altro servizio in un sistema unix-like, viene gestito attraverso un apposito programma, che è anch'esso un *demone*. In generale i messaggi di errore vengono raccolti dal file speciale `/dev/log`, un socket locale (vedi sez. 14.2.4) dedicato a questo scopo, o via rete, con un socket UDP e trattati dal demone che gestisce il servizio. Il più comune di questi è `syslogd`, che consente un semplice smistamento dei messaggi sui file in base alle informazioni in esse presenti; oggi però `syslogd` è in sostanziale disuso, sostituito da programmi più sofisticati come `rsyslog` o `syslog-ng`.

Il servizio del *syslog* permette infatti di trattare i vari messaggi classificandoli attraverso due indici: il primo, chiamato *facility*, suddivide in diverse categorie i messaggi in modo di raggruppare quelli provenienti da operazioni che hanno attinenza fra loro, ed è organizzato in sottosistemi (kernel, posta elettronica, demoni di stampa, ecc.). Il secondo, chiamato *priority*, identifica l'importanza dei vari messaggi, e permette di classificarli e differenziare le modalità di notifica degli stessi.

Il sistema del *syslog* attraverso il proprio demone di gestione provvede poi a riportare i messaggi all'amministratore attraverso una serie differenti meccanismi come:

- scriverli su un file (comunemente detto *log file*, o giornale),
- scriverli sulla console,
- scriverli sui terminali degli utenti connessi,
- inviarli via mail ad uno specifico utente,
- inviarli ad un altro programma,
- inviarli via rete ad una macchina di raccolta,
- ignorarli completamente;

le modalità con cui queste azioni vengono realizzate dipendono ovviamente dal demone che si usa, per la gestione del quale si rimanda ad un testo di amministrazione di sistema.¹³

La *glibc* definisce una serie di funzioni standard con cui un processo può accedere in maniera generica al servizio di *syslog*, che però funzionano solo localmente; se si vogliono

¹³l'argomento è ad esempio coperto dal capitolo 3.2.3 si [?].

inviare i messaggi ad un altro sistema occorre farlo esplicitamente con un socket UDP, o utilizzare le capacità di reinvio del servizio.

La prima funzione definita dall'interfaccia è `openlog`, che inizializza una connessione al servizio di *syslog*. Essa in generale non è necessaria per l'uso del servizio, ma permette di impostare alcuni valori che controllano gli effetti delle chiamate successive; il suo prototipo è:

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility)
                                     Inizializza una connessione al sistema del syslog.
```

La funzione non restituisce nulla.

La funzione permette di specificare, tramite `ident`, l'identità di chi ha inviato il messaggio (di norma si passa il nome del programma, come specificato da `argv[0]`), e la stringa verrà preposta all'inizio di ogni messaggio. Si tenga presente che il valore di `ident` che si passa alla funzione è un puntatore, se la stringa cui punta viene cambiata lo sarà pure nei successivi messaggi, e se viene cancellata i risultati potranno essere imprevedibili, per questo è sempre opportuno usare una stringa costante.

L'argomento `facility` permette invece di preimpostare per le successive chiamate l'omonimo indice che classifica la categoria del messaggio. L'argomento è interpretato come una maschera binaria, e pertanto è possibile inviare i messaggi su più categorie alla volta. I valori delle costanti che identificano ciascuna categoria sono riportati in tab. 8.1, il valore di `facility` deve essere specificato con un OR aritmetico.

Valore	Significato
LOG_AUTH	Messaggi relativi ad autenticazione e sicurezza, obsoleto, è sostituito da LOG_AUTHPRIV.
LOG_AUTHPRIV	Sostituisce LOG_AUTH.
LOG_CRON	Messaggi dei demoni di gestione dei comandi programmati (cron e at).
LOG_DAEMON	Demoni di sistema.
LOG_FTP	Servizio FTP.
LOG_KERN	Messaggi del kernel.
LOG_LOCAL0	Riservato all'amministratore per uso locale.
⋮	⋮
LOG_LOCAL7	Riservato all'amministratore per uso locale.
LOG_LPR	Messaggi del sistema di gestione delle stampanti.
LOG_MAIL	Messaggi del sistema di posta elettronica.
LOG_NEWS	Messaggi del sistema di gestione delle news (USENET).
LOG_SYSLOG	Messaggi generati dal demone di gestione del <i>syslog</i> .
LOG_USER	Messaggi generici a livello utente.
LOG_UUCP	Messaggi del sistema UUCP (<i>Unix to Unix CoPy</i>), ormai in disuso.

Tabella 8.1: Valori possibili per l'argomento `facility` di `openlog`.

L'argomento `option` serve invece per controllare il comportamento della funzione `openlog` e delle modalità con cui le successive chiamate scriveranno i messaggi, esso viene specificato come maschera binaria composta con un OR aritmetico di una qualunque delle costanti riportate in tab. 8.2.

La funzione che si usa per generare un messaggio è `syslog`, dato che l'uso di `openlog` è opzionale, sarà quest'ultima a provvede a chiamare la prima qualora ciò non sia stato fatto (nel qual caso il valore di `ident` è NULL). Il suo prototipo è:

Valore	Significato
LOG_CONS	Scriva sulla console in caso di errore nell'invio del messaggio al sistema del <i>syslog</i> .
LOG_NDELAY	Apri la connessione al sistema del <i>syslog</i> subito invece di attendere l'invio del primo messaggio.
LOG_NOWAIT	Non usato su Linux, su altre piattaforme non attende i processi figli creati per inviare il messaggio.
LOG_ODELAY	Attende il primo messaggio per aprire la connessione al sistema del <i>syslog</i> .
LOG_PERROR	Stampa anche su <i>stderr</i> (non previsto in POSIX.1-2001).
LOG_PID	Inserisce nei messaggi il <i>PID</i> del processo chiamante.

Tabella 8.2: Valori possibili per l'argomento *option* di *openlog*.

```
#include <syslog.h>
void syslog(int priority, const char *format, ...)
Genera un messaggio per il syslog.
```

La funzione non restituisce nulla.

La funzione genera un messaggio le cui caratteristiche sono indicate da *priority*. Per i restanti argomenti il suo comportamento è analogo a quello di *printf*, e il valore dell'argomento *format* è identico a quello descritto nella pagina di manuale di quest'ultima (per i valori principali si può vedere la trattazione sommaria che se ne è fatto in sez. 5.3.6). L'unica differenza è che la sequenza *%m* viene rimpiazzata dalla stringa restituita da *strerror(errno)*. Gli argomenti seguenti i primi due devono essere forniti secondo quanto richiesto da *format*.

Valore	Significato
LOG_EMERG	Il sistema è inutilizzabile.
LOG_ALERT	C'è una emergenza che richiede intervento immediato.
LOG_CRIT	Si è in una condizione critica.
LOG_ERR	Si è in una condizione di errore.
LOG_WARNING	Messaggio di avvertimento.
LOG_NOTICE	Notizia significativa relativa al comportamento.
LOG_INFO	Messaggio informativo.
LOG_DEBUG	Messaggio di debug.

Tabella 8.3: Valori possibili per l'indice di importanza del messaggio da specificare nell'argomento *priority* di *syslog*.

L'argomento *priority* permette di impostare sia la *facility* che la *priority* del messaggio. In realtà viene prevalentemente usato per specificare solo quest'ultima in quanto la prima viene di norma preimpostata con *openlog*. La priorità è indicata con un valore numerico specificabile attraverso le costanti riportate in tab. 8.3.

La *glibc*, seguendo POSIX.1-2001, prevede otto diverse priorità ordinate da 0 a 7, in ordine di importanza decrescente; questo comporta che i tre bit meno significativi dell'argomento *priority* sono occupati da questo valore, mentre i restanti bit più significativi vengono usati per specificare la *facility*. Nel caso si voglia specificare anche la *facility* basta eseguire un OR aritmetico del valore della priorità con la maschera binaria delle costanti di tab. 8.1.

Una funzione sostanzialmente identica a *syslog* è *vsyslog*. La funzione è originaria di BSD e per utilizzarla deve essere definito *_BSD_SOURCE*; il suo prototipo è:

```
#include <syslog.h>
void vsyslog(int priority, const char *format, va_list src)
Genera un messaggio per il syslog.
```

La funzione non restituisce nulla.

La sola differenza con `syslog` è quella di prendere invece di una lista di argomenti esplicita un unico argomento finale passato nella forma di una `va_list`; la funzione risulta utile qualora si ottengano gli argomenti dalla invocazione di un'altra funzione *variadic* (si ricordi quanto visto in sez. 2.4.2).

Per semplificare la gestione della scelta del livello di priorità a partire dal quale si vogliono registrare i messaggi, le funzioni di gestione mantengono per ogni processo una maschera che determina quale delle chiamate effettuate a `syslog` verrà effettivamente registrata. In questo modo sarà possibile escludere i livelli di priorità che non interessa registrare, impostando opportunamente la maschera una volta per tutte.

Questo significa che in genere nei programmi vengono comunque previste le chiamate a `syslog` per tutti i livelli di priorità, ma poi si imposta questa maschera per registrare solo quello che effettivamente interessa. La funzione che consente di fare questo è `setlogmask`, ed il suo prototipo è:

```
#include <syslog.h>
int setlogmask(int mask)
```

Imposta la maschera dei messaggi del *syslog*.

La funzione ritorna il precedente valore della maschera dei messaggi e non prevede errori.

La funzione restituisce il valore della maschera corrente, e se si passa un valore nullo per `mask` la maschera corrente non viene modificata; in questo modo si può leggere il valore della maschera corrente. Indicando un valore non nullo per `mask` la registrazione dei messaggi viene disabilitata per tutte quelle priorità che non rientrano nella maschera.

In genere il valore viene impostato usando la macro `LOG_MASK(p)` dove `p` è una delle costanti di tab. 8.3. È inoltre disponibile anche la macro `LOG_UPTO(p)` che permette di specificare automaticamente tutte le priorità fino a quella indicata da `p`.

Una volta che si sia certi che non si intende registrare più nessun messaggio si può chiudere esplicitamente la connessione al *syslog* (l'uso di questa funzione è comunque completamente opzionale) con la funzione `closelog`, il cui prototipo è:

```
#include <syslog.h>
void closelog(void)
```

Chiude la connessione al *syslog*.

La funzione non ritorna nulla.

Come si evince anche dalla presenza della facility `LOG_KERN` in tab. 8.1, uno dei possibili utenti del servizio del *syslog* è anche il kernel, che a sua volta può avere necessità di inviare messaggi verso l'*user space*. I messaggi del kernel sono mantenuti in un apposito buffer circolare e generati all'interno del kernel tramite la funzione `printk`, analoga alla `printf` usata in *user space*.¹⁴

Come per i messaggi ordinari anche i messaggi del kernel hanno una priorità ma in questo caso non si può contare sulla coincidenza con le costanti di tab. 8.3 dato che il codice del kernel viene mantenuto in maniera indipendente dalle librerie del C. Per questo motivo le varie priorità usate dal kernel sono associate ad un valore numerico che viene tradotto in una stringa preposta ad ogni messaggio, secondo i valori che si sono riportati in fig. 8.2

¹⁴una trattazione eccellente dell'argomento si trova nel quarto capitolo di [?].

```

#define KERN_EMERG      "<0>" /* system is unusable */
#define KERN_ALERT      "<1>" /* action must be taken immediately */
#define KERN_CRIT       "<2>" /* critical conditions */
#define KERN_ERR        "<3>" /* error conditions */
#define KERN_WARNING    "<4>" /* warning conditions */
#define KERN_NOTICE     "<5>" /* normal but significant condition */
#define KERN_INFO       "<6>" /* informational */
#define KERN_DEBUG      "<7>" /* debug-level messages */

```

Figura 8.2: Definizione delle stringhe coi relativi valori numerici che indicano le priorità dei messaggi del kernel (ripresa da `include/linux/kernel.h`).

Dato che i messaggi generati da `printk` hanno un loro specifico formato tradizionalmente si usava un demone ausiliario, `klogd`, per leggerli, rimappare le loro priorità sui valori di tab. 8.3 ed inviarli al sistema del *syslog* nella facility `LOG_KERN`. Oggi i nuovi demoni più avanzati che realizzano il servizio (come `rsyslog` o `syslog-ng`) sono in grado di fare tutto questo da soli leggendoli direttamente senza necessità di un intermediario.

Ma i messaggi del kernel non sono necessariamente connessi al sistema del *syslog*; ad esempio possono anche essere letti direttamente dal buffer circolare con il comando `dmesg`. Inoltre è previsto che se superano una certa priorità essi vengano stampati direttamente sul terminale indicato come *console* di sistema,¹⁵ in modo che sia possibile vederli anche in caso di blocco totale del sistema (nell'assunzione che la console sia collegata).

In particolare la stampa dei messaggi sulla console è controllata dal contenuto del file `/proc/sys/kernel/printk` (o con l'equivalente parametro di `sysctl`) che prevede quattro valori numerici interi: il primo, *console_loglevel*, indica la priorità corrente oltre la quale vengono stampati i messaggi sulla console, il secondo, *default_message_loglevel*, la priorità di default assegnata ai messaggi che non ne hanno impostata una, il terzo, *minimum_console_level*, il valore minimo che si può assegnare al primo valore,¹⁶ ed il quarto, *default_console_loglevel*, il valore di default.¹⁷

Per la lettura dei messaggi del kernel e la gestione del relativo buffer circolare esiste una apposita *system call* chiamata anch'essa `syslog`, ma dato il conflitto di nomi questa viene rimappata su un'altra funzione di libreria, in particolare nella *glibc* essa viene invocata tramite la funzione `klogctl`,¹⁸ il cui prototipo è:

```
#include <sys/klog.h>
int klogctl(int op, char *buffer, int len)
```

Gestisce i messaggi di log del kernel.

La funzione ritorna un intero positivo o nullo dipendente dall'operazione scelta in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL l'argomento `op` non ha un valore valido, o si sono specificati valori non validi per gli altri argomenti quando questi sono richiesti.

ENOSYS il supporto per `printk` non è stato compilato nel kernel.

EPERM non si hanno i privilegi richiesti per l'operazione richiesta.

ERESTARTSYS l'operazione è stata interrotta da un segnale.

¹⁵quello che viene indicato con il parametro di avvio console del kernel, si consulti al riguardo sez. 5.3.1 di [?].

¹⁶che può essere usato con una delle operazioni di gestione che vedremo a breve per "silenziare" il kernel.

¹⁷anch'esso viene usato nelle operazioni di controllo per tornare ad un valore predefinito.

¹⁸nella *libc4* e nella *libc5* la funzione invece era `SYS_klog`.

La funzione prevede che si passi come primo argomento **op** un codice numerico che indica l'operazione richiesta, il secondo argomento deve essere, per le operazioni che compiono una lettura di dati, l'indirizzo del buffer su cui copiarli, ed il terzo quanti byte leggere. L'effettivo uso di questi due argomenti dipende comunque dall'operazione richiesta, ma essi devono essere comunque specificati, anche quando non servono, nel qual caso verranno semplicemente ignorati.

Valore	Significato
0	apre il log (attualmente non fa niente), buffer e len sono ignorati.
1	chiude il log (attualmente non fa niente), buffer e len sono ignorati.
2	legge len byte nel buffer buffer dal log dei messaggi.
3	legge len byte nel buffer buffer dal buffer circolare dei messaggi.
4	legge len byte nel buffer buffer dal buffer circolare dei messaggi e lo svuota.
5	svuota il buffer circolare dei messaggi, buffer e len sono ignorati.
6	disabilita la stampa dei messaggi sulla console, buffer e len sono ignorati.
7	abilita la stampa dei messaggi sulla console, buffer e len sono ignorati.
8	imposta a len il livello dei messaggi stampati sulla console, buffer è ignorato.
9	ritorna il numero di byte da leggere presenti sul buffer di log, buffer e len sono ignorati (dal kernel 2.4.10).
10	ritorna la dimensione del buffer di log, buffer e len sono ignorati (dal kernel 2.6.6).

Tabella 8.4: Valori possibili per l'argomento **op** di **klogctl**.

Si sono riportati in tab. 8.4 i valori utilizzabili per **op**, con una breve spiegazione della relativa operazione e di come vengono usati gli altri due argomenti. Come si può notare la funzione è una sorta di interfaccia comune usata per eseguire operazioni completamente diverse fra loro.

L'operazione corrispondente al valore 2 di **op** consente di leggere un messaggio dal cosiddetto *log* del kernel. Eseguire questa operazione è equivalente ad eseguire una lettura dal file `/proc/kmsg`,¹⁹ se non vi sono messaggi la funzione si blocca in attesa di dati e ritorna soltanto quando questi diventano disponibili. In tal caso verranno letti ed estratti²⁰ dal log **len** byte che verranno scritti su **buffer**; in questo caso il valore di ritorno di **klogctl** corrisponderà al numero di byte ottenuti.

Se invece si usa l'operazione 3 i dati vengono letti dal buffer circolare usato da **printk**, che mantiene tutti i messaggi stampati dal kernel fino al limite delle sue dimensioni, in questo caso i messaggi possono essere letti più volte. Usando invece l'operazione 4 si richiede di cancellare il buffer dopo la lettura, che così risulterà vuoto ad una lettura successiva. Anche con queste operazioni **len** indica il numero di byte da leggere e **buffer** l'indirizzo dove leggerli, e la funzione ritorna il numero di byte effettivamente letti. L'operazione 5 esegue soltanto la cancellazione del buffer circolare, **len** e **buffer** sono ignorati e la funzione ritorna un valore nullo.

Le operazioni corrispondenti ai valori 6, 7 ed 8 consentono di modificare la priorità oltre la quale i messaggi vengono stampati direttamente sulla *console* e fanno riferimento ai parametri del kernel gestiti con le variabili contenute in `/proc/sys/kernel/printk` di cui abbiamo parlato prima, ed in particolare con 6 si imposta come corrente il valore minimo della terza variabile (*minimum_console_level*), ottenendo l'effetto di ridurre al minimo i messaggi che arrivano in

¹⁹in realtà è vero l'opposto, è questa funzione che viene eseguita quando si legge da questo file.

²⁰estratti in quanti i dati del *log* del kernel si possono leggere una volta sola, se più processi eseguono l'operazione di lettura soltanto uno riceverà i dati, a meno che completata la propria operazione di lettura non restino altri messaggi pendenti che a questo punto potrebbero essere letti da un altro processo in attesa.

console, mentre con 7 si ripristina il valore di default.²¹ Per impostare direttamente un valore specifico infine si può usare 8, nel qual caso il valore numerico del livello da impostare deve essere specificato con `len`, che può assumere solo un valore fra 1 e 8.

Infine le due operazioni 9 e 10 consentono di ottenere rispettivamente il numero di byte ancora non letti dal log del kernel, e la dimensione totale di questo. Per entrambe i dati sono restituiti come valore di ritorno, e gli argomenti `buffer` e `len` sono ignorati.

Si tenga presente che la modifica del livello minimo per cui i messaggi vengono stampati sulla console (operazioni 6, 7 e 8) e la cancellazione del buffer circolare di `printk` (operazioni 4 e 5) sono privilegiate; fino al kernel 2.6.30 era richiesta la capacità `CAP_SYS_ADMIN`, a partire dal 2.6.38 detto privilegio è stato assegnato ad una capacità aggiuntiva, `CAP_SYSLOG`. Tutto questo è stato fatto per evitare che processi eseguiti all'interno di un sistema di virtualizzazione “leggera” (come i *Linux Container* di LXC) che necessitano di `CAP_SYS_ADMIN` per operare all'interno del proprio ambiente ristretto, potessero anche avere la capacità di influire sui log del kernel al di fuori di questo.

8.2 L'I/O su terminale

Benché come ogni altro dispositivo i terminali siano accessibili come file, essi hanno assunto storicamente, essendo stati a lungo l'unico modo di accedere al sistema, una loro rilevanza specifica, che abbiamo già avuto modo di incontrare nella precedente sezione.

Esamineremo qui le peculiarità dell'I/O eseguito sui terminali, che per la loro particolare natura presenta delle differenze rispetto ai normali file su disco e agli altri dispositivi.

8.2.1 L'architettura dell'I/O su terminale

I terminali sono una classe speciale di dispositivi a caratteri (si ricordi la classificazione di sez. 1.2.3). Un terminale ha infatti una caratteristica che lo contraddistingue da un qualunque altro dispositivo, è infatti destinato a gestire l'interazione con un utente e deve perciò essere in grado di fare da terminale di controllo per una sessione; tutto questo comporta la presenza di una serie di capacità specifiche.

L'interfaccia per i terminali è una delle più oscure e complesse, essendosi stratificata dagli inizi dei sistemi Unix fino ad oggi. Questo comporta una grande quantità di opzioni e controlli relativi ad un insieme di caratteristiche (come ad esempio la velocità della linea) necessarie per dispositivi, come i terminali seriali, che al giorno d'oggi sono praticamente in disuso.

Storicamente i primi terminali erano appunto terminali di telescriventi (*teletype*), da cui deriva sia il nome dell'interfaccia, *TTY*, che quello dei relativi file di dispositivo, che sono sempre della forma `/dev/tty*`.²² Oggi essi includono le porte seriali, le console virtuali dello schermo, i terminali virtuali che vengono creati come canali di comunicazione dal kernel e che di solito vengono associati alle connessioni di rete (ad esempio per trattare i dati inviati con `telnet` o `ssh`).

L'I/O sui terminali si effettua con le stesse modalità dei file normali: si apre il relativo file di dispositivo, e si leggono e scrivono i dati con le usuali funzioni di lettura e scrittura. Così se apriamo una console virtuale avremo che `read` leggerà quanto immesso dalla tastiera,

²¹secondo la documentazione questo sarebbe quello indicato della quarta variabile, `default_console_loglevel` in genere pari a 7, ma alcune prove con il programma `mydmesg` che si trova nei sorgenti allegati alla guida rivelano che l'unico effetto di questa operazione è riportare il valore a quello precedente se lo si è ridotto al minimo con l'operazione 6.

²²ciò vale solo in parte per i terminali virtuali, essi infatti hanno due lati, un *master*, che può assumere i nomi `/dev/pty[p-za-e][0-9a-f]` ed un corrispondente *slave* con nome `/dev/tty[p-za-e][0-9a-f]`.

mentre `write` scriverà sullo schermo. In realtà questo è vero solo a grandi linee, perché non tiene conto delle caratteristiche specifiche dei terminali; una delle principali infatti è che essi prevedono due modalità di operazione, dette rispettivamente “*modo canonico*” e “*modo non canonico*”, che hanno dei comportamenti nettamente diversi.

La modalità preimpostata all'apertura del terminale è quella canonica, in cui le operazioni di lettura vengono sempre effettuate assemblando i dati in una linea. Questo significa che eseguendo una `read` su un terminale in modo canonico la funzione si bloccherà, anche se si sono scritti dei caratteri, fintanto che non si preme il tasto di ritorno a capo: a questo punto la linea sarà completata e la funzione ritornerà leggendola per intero.

Inoltre in modalità canonica alcuni dei caratteri che si scrivono sul terminale vengono interpretati direttamente dal kernel per compiere operazioni (come la generazione dei segnali associati al *job control* illustrata in sez. 7.2.6), e non saranno mai letti dal dispositivo. Quella canonica è di norma la modalità in cui opera la shell.

Un terminale in modo non canonico invece non effettua nessun accorpamento dei dati in linee né li interpreta; esso viene di solito usato da quei programmi, come ad esempio gli editor, che necessitano di poter leggere un carattere alla volta e che gestiscono al loro interno l'interpretazione dei caratteri ricevuti impiegandoli opportunamente come comandi o come dati.

Figura 8.3: Struttura interna generica del kernel per l'accesso ai dati di un terminale.

Per capire le caratteristiche dell'I/O sui terminali occorre esaminare le modalità con cui esso viene effettuato. L'accesso, come per tutti i dispositivi, viene gestito dal kernel, ma per tutti i terminali viene utilizzata una architettura generica che si è schematizzata in fig. 8.3.

Ad ogni terminale sono sempre associate due code per gestire l'input e l'output, che ne implementano una bufferizzazione all'interno del kernel che è completamente indipendente dalla eventuale ulteriore bufferizzazione fornita dall'interfaccia standard dei file.

La coda di ingresso mantiene i caratteri che sono stati letti dal terminale ma non ancora letti da un processo, la sua dimensione è definita dal parametro di sistema `MAX_INPUT` (si veda sez. 6.1.2), che ne specifica il limite minimo, in realtà la coda può essere più grande e cambiare dimensione dinamicamente.

Se è stato abilitato il controllo di flusso in ingresso il kernel emette i caratteri di `STOP` e `START` per bloccare e sbloccare l'ingresso dei dati; altrimenti i caratteri immessi oltre le dimensioni massime vengono persi; in alcuni casi il kernel provvede ad inviare automaticamente un avviso (un carattere di `BELL`, che provoca un beep) sull'output quando si eccedono le dimensioni della coda.

Se è abilitato la modalità canonica i caratteri in ingresso restano nella coda fintanto che non viene ricevuto un a capo; un altro parametro del sistema, `MAX_CANON`, specifica la dimensione massima di una riga in modalità canonica.

La coda di uscita è analoga a quella di ingresso e contiene i caratteri scritti dai processi ma non ancora inviati al terminale. Se è abilitato il controllo di flusso in uscita il kernel risponde ai caratteri di `START` e `STOP` inviati dal terminale. Le dimensioni della coda non sono specificate, ma non hanno molta importanza, in quanto qualora esse vengano eccedute il driver provvede automaticamente a bloccare la funzione chiamante.

8.2.2 La gestione delle caratteristiche di un terminale

Data le loro peculiarità, fin dalla realizzazione dei primi sistemi unix-like si è posto il problema di come gestire le caratteristiche specifiche dei terminali. Storicamente i vari dialetti di Unix

hanno utilizzato delle funzioni specifiche diverse fra loro, ma con la realizzazione dello standard POSIX.1-2001 è stata effettuata opportuna unificazione delle funzioni attinenti i terminali, sintetizzando le differenze fra BSD e System V in una unica interfaccia, che è quella adottata da Linux.

Molte delle funzioni previste dallo standard POSIX.1-2001 prevedono come argomento un file descriptor, dato che in origine le relative operazioni venivano effettuate con delle opportune chiamate a `ioctl`. Ovviamente dette funzioni potranno essere usate correttamente soltanto con dei file descriptor che corrispondono ad un terminale, in caso contrario lo standard richiede che venga restituito un errore di `ENOTTY`.

Per evitare l'errore, ed anche semplicemente per verificare se un file descriptor corrisponde ad un terminale, è disponibile la funzione `isatty`, il cui prototipo è:

<pre>#include <unistd.h> int isatty(int fd)</pre>	Controlla se un file è un terminale.
La funzione ritorna 1 se <code>fd</code> è connesso ad un terminale e 0 altrimenti, nel qual caso <code>errno</code> assumerà uno dei valori:	
<code>EBADF</code> <code>fd</code> non è un file descriptor valido.	
<code>EINVAL</code> <code>fd</code> non è associato a un terminale (non ottempera a POSIX.1-2001 che richiederebbe <code>ENOTTY</code>).	

Un'altra funzione per avere informazioni su un terminale è `ttyname`, che permette di ottenere il nome del file di dispositivo del terminale associato ad un file descriptor; il suo prototipo è:

<pre>#include <unistd.h> char *ttyname(int fd)</pre>	Restituisce il nome del terminale associato ad un file descriptor.
La funzione ritorna il puntatore alla stringa contenente il nome del terminale in caso di successo e <code>NULL</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
<code>EBADF</code> <code>fd</code> non è un file descriptor valido.	
<code>ENOTTY</code> <code>fd</code> non è associato a un terminale.	

La funzione restituisce il puntatore alla stringa contenente il nome del file di dispositivo del terminale associato a `fd`, che però è allocata staticamente e può essere sovrascritta da successive chiamate. Per questo della funzione esiste anche una versione rientrante, `ttyname_r`, che non presenta il problema dell'uso di una zona di memoria statica; il suo prototipo è:

<pre>#include <unistd.h> int ttyname_r(int fd, char *buff, size_t len)</pre>	Restituisce il nome del terminale associato ad un file descriptor.
La funzione ritorna 0 in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
<code>ERANGE</code> la lunghezza del buffer <code>len</code> non è sufficiente per contenere la stringa restituita.	
ed inoltre <code>EBADF</code> ed <code>ENOTTY</code> con lo stesso significato di <code>ttyname</code> .	

La funzione prende due argomenti in più, il puntatore `buff` alla zona di memoria in cui l'utente vuole che il risultato venga scritto, che dovrà essere allocata in precedenza, e la relativa dimensione, `len`. Se la stringa che deve essere restituita, compreso lo zero di terminazione finale, eccede questa dimensione si avrà un errore.

Una funzione analoga alle precedenti prevista da POSIX.1, che restituisce il nome di un file di dispositivo, è `ctermid`, il cui prototipo è:

```
#include <stdio.h>
char *ctermid(char *s)
```

Restituisce il nome del terminale di controllo del processo.

La funzione ritorna il puntatore alla stringa contenente il *pathname* del terminale o NULL se non riesce ad eseguire l'operazione, non sono previsti errori.

La funzione restituisce un puntatore al *pathname* del file di dispositivo del terminale di controllo del processo chiamante. Se si passa come argomento NULL la funzione restituisce il puntatore ad una stringa statica che può essere sovrascritta da chiamate successive, e non è rientrante. Indicando invece un puntatore ad una zona di memoria già allocata la stringa sarà scritta su di essa, ma in questo caso il buffer preallocato deve avere una dimensione di almeno `L_ctermid` caratteri.²³

Si tenga presente che il *pathname* restituito dalla funzione potrebbe non identificare univocamente il terminale (ad esempio potrebbe essere `/dev/tty`), inoltre non è detto che il processo possa effettivamente essere in grado di aprire il terminale.

I vari attributi associati ad un terminale vengono mantenuti per ciascuno di essi in una struttura `termios` che viene usata dalle varie funzioni dell'interfaccia. In fig. 8.4 si sono riportati tutti i campi della definizione di questa struttura usata in Linux; di questi solo i primi cinque sono previsti dallo standard POSIX.1, ma le varie implementazioni ne aggiungono degli altri per mantenere ulteriori informazioni.²⁴

```
struct termios {
    tcflag_t c_iflag;      /* input mode flagss */
    tcflag_t c_oflag;      /* output modes flags */
    tcflag_t c_cflag;      /* control modes flags */
    tcflag_t c_lflag;      /* local modes flags */
    cc_t c_line;           /* line discipline */
    cc_t c_cc[NCCS];       /* control characters */
    speed_t c_ispeed;      /* input speed */
    speed_t c_ospeed;      /* output speed */
};
```

Figura 8.4: La struttura `termios`, che identifica le proprietà di un terminale.

I primi quattro campi sono quattro flag che controllano il comportamento del terminale; essi sono realizzati come maschera binaria, pertanto il tipo `tcflag_t` è di norma realizzato con un intero senza segno di lunghezza opportuna. I valori devono essere specificati bit per bit, avendo cura di non modificare i bit su cui non si interviene.

Il primo flag, mantenuto nel campo `c_iflag`, è detto *flag di input* e controlla le modalità di funzionamento dell'input dei caratteri sul terminale, come il controllo di parità, il controllo di flusso, la gestione dei caratteri speciali; un elenco dei vari bit, del loro significato e delle costanti utilizzate per identificarli è riportato in tab. 8.5.

Si noti come alcuni di questi flag (come quelli per la gestione del flusso) fanno riferimento a delle caratteristiche che ormai sono completamente obsolete; la maggior parte inoltre è tipica di terminali seriali, e non ha alcun effetto su dispositivi diversi come le console virtuali o gli pseudo-terminali usati nelle connessioni di rete.

²³`L_ctermid` è una delle varie costanti del sistema, non trattata esplicitamente in sez. 6.1, che indica la dimensione che deve avere una stringa per poter contenere il nome di un terminale.

²⁴la definizione della struttura si trova in `bits/termios.h`, da non includere mai direttamente; Linux, seguendo l'esempio di BSD, aggiunge i due campi `c_ispeed` e `c_ospeed` per mantenere le velocità delle linee seriali, ed un campo ulteriore, `c_line` per indicare la disciplina di linea.

Valore	Significato
IGNBRK	Ignora le condizioni di BREAK sull'input. Una <i>condizione di BREAK</i> è definita nel contesto di una trasmissione seriale asincrona come una sequenza di bit nulli più lunga di un byte.
BRKINT	Controlla la reazione ad un BREAK quando IGNBRK non è impostato. Se BRKINT è impostato il BREAK causa lo scarico delle code, e se il terminale è il terminale di controllo per un gruppo in foreground anche l'invio di SIGINT ai processi di quest'ultimo. Se invece BRKINT non è impostato un BREAK viene letto come un carattere NUL, a meno che non sia impostato PARMRK nel qual caso viene letto come la sequenza di caratteri 0xFF 0x00 0x00.
IGNPAR	Ignora gli errori di parità, il carattere viene passato come ricevuto. Ha senso solo se si è impostato INPCK.
PARMRK	Controlla come vengono riportati gli errori di parità. Ha senso solo se INPCK è impostato e IGNPAR no. Se impostato inserisce una sequenza 0xFF 0x00 prima di ogni carattere che presenta errori di parità, se non impostato un carattere con errori di parità viene letto come uno 0x00. Se un carattere ha il valore 0xFF e ISTRIP non è impostato, per evitare ambiguità esso viene sempre riportato come 0xFF 0xFF.
INPCK	Abilita il controllo di parità in ingresso. Se non viene impostato non viene fatto nessun controllo ed i caratteri vengono passati in input direttamente.
ISTRIP	Se impostato i caratteri in input sono tagliati a sette bit mettendo a zero il bit più significativo, altrimenti vengono passati tutti gli otto bit.
INLCR	Se impostato in ingresso il carattere di a capo ('\n') viene automaticamente trasformato in un ritorno carrello ('\r').
IGNCR	Se impostato il carattere di ritorno carrello (<i>carriage return</i> , '\r') viene scaricato dall'input. Può essere utile per i terminali che inviano entrambi i caratteri di ritorno carrello e a capo (<i>newline</i> , '\n').
ICRNL	Se impostato un carattere di ritorno carrello ('\r') sul terminale viene automaticamente trasformato in un a capo ('\n') sulla coda di input.
IUCLC	Se impostato trasforma i caratteri maiuscoli dal terminale in minuscoli sull'ingresso (opzione non POSIX).
IXON	Se impostato attiva il controllo di flusso in uscita con i caratteri di START e STOP. se si riceve uno STOP l'output viene bloccato, e viene fatto ripartire solo da uno START, e questi due caratteri non vengono passati alla coda di input. Se non impostato i due caratteri sono passati alla coda di input insieme agli altri.
IXANY	Se impostato con il controllo di flusso permette a qualunque carattere di far ripartire l'output bloccato da un carattere di STOP.
IXOFF	Se impostato abilita il controllo di flusso in ingresso. Il computer emette un carattere di STOP per bloccare l'input dal terminale e lo sblocca con il carattere START.
IMAXBEL	Se impostato fa suonare il cicalino se si riempie la coda di ingresso; in Linux non è implementato e il kernel si comporta come se fosse sempre impostato (è una estensione BSD).
IUTF8	Indica che l'input è in UTF-8, cosa che consente di utilizzare la cancellazione dei caratteri in maniera corretta (dal kernel 2.6.4 e non previsto in POSIX).

Tabella 8.5: Costanti identificative dei vari bit del flag di controllo `c_iflag` delle modalità di input di un terminale.

Il secondo flag, mantenuto nel campo `c_oflag`, è detto *flag di output* e controlla le modalità di funzionamento dell'output dei caratteri, come l'impacchettamento dei caratteri sullo schermo, la traslazione degli a capo, la conversione dei caratteri speciali; un elenco dei vari bit, del loro significato e delle costanti utilizzate per identificarli è riportato in tab. 8.6, di questi solo `OPOST` era previsto da POSIX.1, buona parte degli altri sono stati aggiunti in POSIX.1-2001, quelli ancora assenti sono stati indicati esplicitamente.

Si noti come alcuni dei valori riportati in tab. 8.6 fanno riferimento a delle maschere di bit; essi infatti vengono utilizzati per impostare alcuni valori numerici relativi ai ritardi

Valore	Significato
OPOST	Se impostato i caratteri vengono convertiti opportunamente (in maniera dipendente dall'implementazione) per la visualizzazione sul terminale, ad esempio al carattere di a capo (NL) può venire aggiunto un ritorno carrello (CR).
OLCUC	Se impostato trasforma i caratteri minuscoli in ingresso in caratteri maiuscoli sull'uscita (non previsto da POSIX).
ONLCR	Se impostato converte automaticamente il carattere di a capo (NL) in un carattere di ritorno carrello (CR).
OCRNL	Se impostato converte automaticamente il carattere di a capo (NL) nella coppia di caratteri ritorno carrello, a capo (CR-NL).
ONOCR	Se impostato converte il carattere di ritorno carrello (CR) nella coppia di caratteri CR-NL.
ONLRET	Se impostato rimuove dall'output il carattere di ritorno carrello (CR).
OFILL	Se impostato in caso di ritardo sulla linea invia dei caratteri di riempimento invece di attendere.
OFDEL	Se impostato il carattere di riempimento è DEL (0x3F), invece che NUL (0x00), (non previsto da POSIX e non implementato su Linux).
NLDLY	Maschera per i bit che indicano il ritardo per il carattere di a capo (NL), i valori possibili sono NL0 o NL1.
CRDLY	Maschera per i bit che indicano il ritardo per il carattere ritorno carrello (CR), i valori possibili sono CR0, CR1, CR2 o CR3.
TABDLY	Maschera per i bit che indicano il ritardo per il carattere di tabulazione, i valori possibili sono TAB0, TAB1, TAB2 o TAB3.
BSDLY	Maschera per i bit che indicano il ritardo per il carattere di ritorno indietro (<i>backspace</i>), i valori possibili sono BS0 o BS1.
VTDLY	Maschera per i bit che indicano il ritardo per il carattere di tabulazione verticale, i valori possibili sono VT0 o VT1.
FFDLY	Maschera per i bit che indicano il ritardo per il carattere di pagina nuova (<i>form feed</i>), i valori possibili sono FF0 o FF1.

Tabella 8.6: Costanti identificative dei vari bit del flag di controllo `c_oflag` delle modalità di output di un terminale.

nell'output di alcuni caratteri: una caratteristica originaria dei primi terminali su telescrivente, che avevano bisogno di tempistiche diverse per spostare il carrello in risposta ai caratteri speciali, e che oggi sono completamente in disuso.

Si tenga presente inoltre che nel caso delle maschere il valore da inserire in `c_oflag` deve essere fornito avendo cura di cancellare prima tutti i bit della maschera, i valori da immettere infatti (quelli riportati nella spiegazione corrispondente) sono numerici e non per bit, per cui possono sovrapporsi fra di loro. Occorrerà perciò utilizzare un codice del tipo:

```
c_oflag &= (~CRDLY);
c_oflag |= CR1;
```

che prima cancella i bit della maschera in questione e poi imposta il valore.

Il terzo flag, mantenuto nel campo `c_cflag`, è detto *flag di controllo* ed è legato al funzionamento delle linee seriali, permettendo di impostarne varie caratteristiche, come il numero di bit di stop, le impostazioni della parità, il funzionamento del controllo di flusso; esso ha senso solo per i terminali connessi a linee seriali. Un elenco dei vari bit, del loro significato e delle costanti utilizzate per identificarli è riportato in tab. 8.7.

I valori previsti per questo flag sono molto specifici, e completamente attinenti al controllo delle modalità operative di un terminale che opera attraverso una linea seriale; essi pertanto non hanno nessuna rilevanza per i terminali che usano un'altra interfaccia fisica, come le console virtuali e gli pseudo-terminali usati dalle connessioni di rete.

Valore	Significato
CBAUD	Maschera dei bit (4+1) usati per impostare della velocità della linea (il <i>baud rate</i>) in ingresso; non è presente in POSIX ed in Linux non è implementato in quanto viene usato un apposito campo di <i>termios</i> .
CBAUDEX	Bit aggiuntivo per l'impostazione della velocità della linea, non è presente in POSIX e per le stesse motivazioni del precedente non è implementato in Linux.
CSIZE	Maschera per i bit usati per specificare la dimensione del carattere inviato lungo la linea di trasmissione, i valore ne indica la lunghezza (in bit), ed i valori possibili sono CS5, CS6, CS7 e CS8 corrispondenti ad un analogo numero di bit.
CSTOPB	Se impostato vengono usati due bit di stop sulla linea seriale, se non impostato ne viene usato soltanto uno.
CREAD	Se è impostato si può leggere l'input del terminale, altrimenti i caratteri in ingresso vengono scartati quando arrivano.
PARENB	Se impostato abilita la generazione il controllo di parità. La reazione in caso di errori dipende dai relativi valori per <i>c_iflag</i> , riportati in tab. 8.5. Se non è impostato i bit di parità non vengono generati e i caratteri non vengono controllati.
PARODD	Ha senso solo se è attivo anche PARENB. Se impostato viene usata una parità è dispari, altrimenti viene usata una parità pari.
HUPCL	Se è impostato viene distaccata la connessione del modem quando l'ultimo dei processi che ha ancora un file aperto sul terminale lo chiude o esce.
LOBLK	Se impostato blocca l'output su un strato di shell non corrente, non è presente in POSIX e non è implementato da Linux.
CLOCAL	Se impostato indica che il terminale è connesso in locale e che le linee di controllo del modem devono essere ignorate. Se non impostato effettuando una chiamata ad open senza aver specificato il flag di <i>O_NONBLOCK</i> si bloccherà il processo finché non si è stabilita una connessione con il modem; inoltre se viene rilevata una disconnessione viene inviato un segnale di <i>SIGHUP</i> al processo di controllo del terminale. La lettura su un terminale sconnesso comporta una condizione di <i>end of file</i> e la scrittura un errore di <i>EIO</i> .
CIBAUD	Maschera dei bit della velocità della linea in ingresso; analogo a CBAUD, non è previsto da POSIX e non è implementato in Linux dato che è mantenuto in un apposito campo di <i>termios</i> .
CMSPAR	imposta un bit di parità costante: se PARODD è impostato la parità è sempre 1 (<i>MARK</i>) se non è impostato la parità è sempre 0 (<i>SPACE</i>), non è previsto da POSIX.
CRTSCTS	Abilita il controllo di flusso hardware sulla seriale, attraverso l'utilizzo delle dei due fili di RTS e CTS.

Tabella 8.7: Costanti identificative dei vari bit del flag di controllo *c_cflag* delle modalità di controllo di un terminale.

Inoltre alcuni valori di questi flag sono previsti solo per quelle implementazioni (lo standard POSIX non specifica nulla riguardo l'implementazione, ma solo delle funzioni di lettura e scrittura) che mantengono le velocità delle linee seriali all'interno dei flag; come accennato in Linux questo viene fatto (seguendo l'esempio di BSD) attraverso due campi aggiuntivi, *c_ispeed* e *c_ospeed*, nella struttura *termios* (mostrati in fig. 8.4).

Il quarto flag, mantenuto nel campo *c_lflag*, è detto *flag locale*, e serve per controllare il funzionamento dell'interfaccia fra il driver e l'utente, come abilitare l'eco, gestire i caratteri di controllo e l'emissione dei segnali, impostare modo canonico o non canonico. Un elenco dei vari bit, del loro significato e delle costanti utilizzate per identificarli è riportato in tab. 8.8. Con i terminali odierni l'unico flag con cui probabilmente si può avere a che fare è questo, in quanto è con questo che si impostano le caratteristiche generiche comuni a tutti i terminali.

Si tenga presente che i flag che riguardano le modalità di eco dei caratteri (*ECHOE*, *ECHOPRT*, *ECHOK*, *ECHOKE*, *ECHONL*) controllano solo il comportamento della visualizzazione, il ricono-

Valore	Significato
ISIG	Se impostato abilita il riconoscimento dei caratteri INTR, QUIT, e SUSP generando il relativo segnale.
ICANON	Se impostato il terminale opera in modalità canonica, altrimenti opera in modalità non canonica.
XCASE	Se impostato il terminale funziona solo con le maiuscole. L'input è convertito in minuscole tranne per i caratteri preceduti da una "\". In output le maiuscole sono precedute da una "\" e le minuscole convertite in maiuscole. Non è presente in POSIX.
ECHO	Se è impostato viene attivato l'eco dei caratteri in input sull'output del terminale.
ECHOE	Se è impostato l'eco mostra la cancellazione di un carattere in input (in reazione al carattere ERASE) cancellando l'ultimo carattere della riga corrente dallo schermo; altrimenti il carattere è rimandato in eco per mostrare quanto accaduto (usato per i terminali con l'uscita su una stampante).
ECHOK	Se impostato abilita il trattamento della visualizzazione del carattere KILL, andando a capo dopo aver visualizzato lo stesso, altrimenti viene solo mostrato il carattere e sta all'utente ricordare che l'input precedente è stato cancellato.
ECHONL	Se impostato viene effettuato l'eco di un a capo (\n) anche se non è stato impostato ECHO.
ECHOCTL	Se impostato insieme ad ECHO i caratteri di controllo ASCII (tranne TAB, NL, START, e STOP) sono mostrati nella forma che prepone un "^" alla lettera ottenuta sommando 0x40 al valore del carattere (di solito questi si possono ottenere anche direttamente premendo il tasto ctrl più la relativa lettera). Non è presente in POSIX.
ECHOPRT	Se impostato abilita la visualizzazione del carattere di cancellazione in una modalità adatta ai terminali con l'uscita su stampante; l'invio del carattere di ERASE comporta la stampa di un " " seguito dal carattere cancellato, e così via in caso di successive cancellazioni, quando si riprende ad immettere carattere normali prima verrà stampata una "/". Non è presente in POSIX.
ECHOKE	Se impostato abilita il trattamento della visualizzazione del carattere KILL cancellando i caratteri precedenti nella linea secondo le modalità specificate dai valori di ECHOE e ECHOPRT. Non è presente in POSIX.
DEFECHO	Se impostato effettua l'eco solo se c'è un processo in lettura. Non è presente in POSIX e non è supportato da Linux.
FLUSHO	Effettua la cancellazione della coda di uscita. Viene attivato dal carattere DISCARD. Non è presente in POSIX e non è supportato da Linux.
NOFLSH	Se impostato disabilita lo scarico delle code di ingresso e uscita quando vengono emessi i segnali SIGINT, SIGQUIT e SIGSUSP.
TOSTOP	Se abilitato, con il supporto per il job control presente, genera il segnale SIGTTOU per un processo in background che cerca di scrivere sul terminale.
PENDIN	Indica che la linea deve essere ristampata, viene attivato dal carattere RE-PRINT e resta attivo fino alla fine della ristampa. Non è presente in POSIX e non è supportato in Linux.
IEXTEN	Abilita alcune estensioni previste dalla implementazione. Deve essere impostato perché caratteri speciali come EOL2, LNEXT, REPRINT e WERASE possano essere interpretati.

Tabella 8.8: Costanti identificative dei vari bit del flag di controllo `c_lflag` delle modalità locali di un terminale.

scimento dei vari caratteri dipende dalla modalità di operazione, ed avviene solo in modo canonico, pertanto questi flag non hanno significato se non è impostato ICANON.

Oltre ai vari flag per gestire le varie caratteristiche dei terminali, `termios` contiene pure il campo `c_cc` che viene usato per impostare i caratteri speciali associati alle varie funzioni di controllo. Il numero di questi caratteri speciali è indicato dalla costante `NCCS`, POSIX ne specifica almeno 11, ma molte implementazioni ne definiscono molti altri.²⁵

²⁵in Linux il valore della costante è 32, anche se i caratteri effettivamente definiti sono solo 17.

Indice	Valore	Codice	Funzione
VINTR	0x03	(C-c)	Carattere di interrupt, provoca l'emissione di SIGINT.
VQUIT	0x1C	(C-\)	Carattere di uscita, provoca l'emissione di SIGQUIT.
VERASE	0x7f	DEL,C-?	Carattere di ERASE, cancella l'ultimo carattere precedente nella linea.
VKILL	0x15	(C-u)	Carattere di KILL, cancella l'intera riga.
VEOF	0x04	(C-d)	Carattere di <i>end-of-file</i> . Causa l'invio del contenuto del buffer di ingresso al processo in lettura anche se non è ancora stato ricevuto un a capo. Se è il primo carattere immesso comporta il ritorno di read con zero caratteri, cioè la condizione di <i>end-of-file</i> .
VMIN	—	—	Numero minimo di caratteri per una lettura in modo non canonico.
VEOL	0x00	NUL	Carattere di fine riga. Agisce come un a capo, ma non viene scartato ed è letto come l'ultimo carattere nella riga.
VTIME	—	—	Timeout, in decimi di secondo, per una lettura in modo non canonico.
VEOL2	0x00	NUL	Ulteriore carattere di fine riga. Ha lo stesso effetto di VEOL ma può essere un carattere diverso.
VSWTC	0x00	NUL	Carattere di switch. Non supportato in Linux.
VSTART	0x17	(C-q)	Carattere di START. Riavvia un output bloccato da uno STOP.
VSTOP	0x19	(C-s)	Carattere di STOP. Blocca l'output fintanto che non viene premuto un carattere di START.
VSUSP	0x1A	(C-z)	Carattere di sospensione. Invia il segnale SIGTSTP.
VDSUSP	0x19	(C-y)	Carattere di sospensione ritardata. Invia il segnale SIGTSTP quando il carattere viene letto dal programma, (non presente in POSIX e non riconosciuto in Linux).
VLNEXT	0x16	(C-v)	Carattere di escape, serve a quotare il carattere successivo che non viene interpretato ma passato direttamente all'output.
VWERASE	0x17	(C-w)	Cancellazione di una parola.
VREPRINT	0x12	(C-r)	Ristampa i caratteri non ancora letti (non presente in POSIX).
VDISCARD	0x0F	(C-o)	Non riconosciuto in Linux.
VSTATUS	0x13	(C-t)	Non riconosciuto in Linux.

Tabella 8.9: Valori dei caratteri di controllo mantenuti nel campo `c_cc` della struttura `termios`.

A ciascuna di queste funzioni di controllo corrisponde un elemento del vettore `c_cc` che specifica quale è il carattere speciale associato; per portabilità invece di essere indicati con la loro posizione numerica nel vettore, i vari elementi vengono indicizzati attraverso delle opportune costanti, il cui nome corrisponde all'azione ad essi associata. Un elenco completo dei caratteri di controllo, con le costanti e delle funzionalità associate è riportato in tab. 8.9, usando quelle definizioni diventa possibile assegnare un nuovo carattere di controllo con un codice del tipo:

```
value.c_cc[VEOL2] = '\n';
```

La maggior parte di questi caratteri (tutti tranne `VTIME` e `VMIN`) hanno effetto solo quando il terminale viene utilizzato in modo canonico; per alcuni devono essere soddisfatte ulteriori richieste, ad esempio `VINTR`, `VSUSP`, e `VQUIT` richiedono sia impostato `ISIG`; `VSTART` e `VSTOP` richiedono sia impostato `IXON`; `VLNEXT`, `VWERASE`, `VREPRINT` richiedono sia impostato `IEXTEN`. In ogni caso quando vengono attivati i caratteri vengono interpretati e non sono passati sulla coda di ingresso.

Per leggere ed scrivere tutte le varie impostazioni dei terminali viste finora lo standard POSIX prevede due funzioni che utilizzano come argomento un puntatore ad una struttura `termios` che sarà quella in cui andranno immagazzinate le impostazioni. Le funzioni sono `tcgetattr` e `tcsetattr` ed il loro prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcgetattr(int fd, struct termios *termios_p)
    Legge il valore delle impostazioni di un terminale.
int tcsetattr(int fd, int optional_actions, struct termios *termios_p)
    Scrive le impostazioni di un terminale.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:
EINTR la funzione è stata interrotta.
 ed inoltre **EBADF**, **ENOTTY** ed **EINVAL** nel loro significato generico.

Le funzioni operano sul terminale cui fa riferimento il file descriptor `fd` utilizzando la struttura indicata dal puntatore `termios_p` per lo scambio dei dati. Si tenga presente che le impostazioni sono associate al terminale e non al file descriptor; questo significa che se si è cambiata una impostazione un qualunque altro processo che apra lo stesso terminale, od un qualunque altro file descriptor che vi faccia riferimento, vedrà le nuove impostazioni pur non avendo nulla a che fare con il file descriptor che si è usato per effettuare i cambiamenti.

Questo significa che non è possibile usare file descriptor diversi per utilizzare automaticamente il terminale in modalità diverse, se esiste una necessità di accesso differenziato di questo tipo occorrerà cambiare esplicitamente la modalità tutte le volte che si passa da un file descriptor ad un altro.

La funzione `tcgetattr` legge i valori correnti delle impostazioni di un terminale qualunque nella struttura puntata da `termios_p`; `tcsetattr` invece effettua la scrittura delle impostazioni e quando viene invocata sul proprio terminale di controllo può essere eseguita con successo solo da un processo in foreground. Se invocata da un processo in background infatti tutto il gruppo riceverà un segnale di **SIGTTOU** come se si fosse tentata una scrittura, a meno che il processo chiamante non abbia **SIGTTOU** ignorato o bloccato, nel qual caso l'operazione sarà eseguita.

La funzione `tcsetattr` prevede tre diverse modalità di funzionamento, specificabili attraverso l'argomento `optional_actions`, che permette di stabilire come viene eseguito il cambiamento delle impostazioni del terminale, i valori possibili sono riportati in tab. 8.10; di norma (come fatto per le due funzioni di esempio) si usa sempre **TCSANOW**, le altre opzioni possono essere utili qualora si cambino i parametri di output.

Valore	Significato
TCSANOW	Esegue i cambiamenti in maniera immediata.
TCSADRAIN	I cambiamenti vengono eseguiti dopo aver atteso che tutto l'output presente sulle code è stato scritto.
TCSAFLUSH	È identico a TCSADRAIN , ma in più scarta tutti i dati presenti sulla coda di input.

Tabella 8.10: Possibili valori per l'argomento `optional_actions` della funzione `tcsetattr`.

Occorre infine tenere presente che `tcsetattr` ritorna con successo anche se soltanto uno dei cambiamenti richiesti è stato eseguito. Pertanto se si effettuano più cambiamenti è buona norma controllare con una ulteriore chiamata a `tcgetattr` che essi siano stati eseguiti tutti quanti.

Come già accennato per i cambiamenti effettuati ai vari flag di controllo occorre che i valori di ciascun bit siano specificati avendo cura di mantenere intatti gli altri; per questo motivo in generale si deve prima leggere il valore corrente delle impostazioni con `tcgetattr` per poi modificare i valori impostati.

```

1 #include <unistd.h>
2 #include <termios.h>
3 #include <errno.h>
4 int SetTermAttr(int fd, tcflag_t flag)
5 {
6     struct termios values;
7     int res;
8     if (res = tcgetattr(desc, &values)) {
9         perror("Cannot_get_attributes");
10        return res;
11    }
12    values.c_lflag |= flag;
13    if (res = tcsetattr(desc, TCSANOW, &values)) {
14        perror("Cannot_set_attributes");
15        return res;
16    }
17    return 0;
18 }

```

Figura 8.5: Codice della funzione `SetTermAttr` che permette di impostare uno dei flag di controllo locale del terminale.

In fig. 8.5 e fig. 8.6 si è riportato rispettivamente il codice delle due funzioni `SetTermAttr` e `UnSetTermAttr`, che possono essere usate per impostare o rimuovere, con le dovute precauzioni, un qualunque bit di `c_lflag`. Il codice completo di entrambe le funzioni può essere trovato nel file `SetTermAttr.c` dei sorgenti allegati alla guida.

La funzione `SetTermAttr` provvede ad impostare il bit specificato dall'argomento `flag`; prima si leggono i valori correnti (8) con `tcgetattr`, uscendo con un messaggio in caso di errore (9-10), poi si provvede a impostare solo i bit richiesti (possono essere più di uno) con un OR binario (12); infine si scrive il nuovo valore modificato con `tcsetattr` (13), notificando un eventuale errore (14-15) o uscendo normalmente.

```

1 int UnSetTermAttr(int fd, tcflag_t flag)
2 {
3     struct termios values;
4     int res;
5     if (res = tcgetattr(desc, &values)) {
6         perror("Cannot_get_attributes");
7         return res;
8     }
9     values.c_lflag &= (~flag);
10    if (res = tcsetattr(desc, TCSANOW, &values)) {
11        perror("Cannot_set_attributes");
12        return res;
13    }
14    return 0;
15 }

```

Figura 8.6: Codice della funzione `UnSetTermAttr` che permette di rimuovere uno dei flag di controllo locale del terminale.

La seconda funzione, `UnSetTermAttr`, è assolutamente identica alla prima, solo che in

questo caso (9) si rimuovono i bit specificati dall'argomento `flag` usando un AND binario del valore negato.

Al contrario di tutte le altre caratteristiche dei terminali, che possono essere impostate esplicitamente utilizzando gli opportuni campi di `termios`, per le velocità della linea (il cosiddetto *baud rate*) non è prevista una implementazione standardizzata, per cui anche se in Linux sono mantenute in due campi dedicati nella struttura, questi non devono essere acceduti direttamente ma solo attraverso le apposite funzioni di interfaccia provviste da POSIX.1.

Lo standard prevede due funzioni per scrivere la velocità delle linee seriali, `cfsetispeed` per la velocità della linea di ingresso e `cfsetospeed` per la velocità della linea di uscita; i loro prototipi sono:

```
#include <unistd.h>
#include <termios.h>
int cfsetispeed(struct termios *termios_p, speed_t speed)
                                Imposta la velocità delle linee seriali in ingresso.
int cfsetospeed(struct termios *termios_p, speed_t speed)
                                Imposta la velocità delle linee seriali in uscita.
```

Le funzioni ritornano 0 in caso di successo e -1 per un errore, che avviene solo quando il valore specificato non è valido.

Si noti che le funzioni si limitano a scrivere opportunamente il valore della velocità prescelta `speed` all'interno della struttura puntata da `termios_p`; per effettuare l'impostazione effettiva occorrerà poi chiamare `tcsetattr`.

Si tenga presente che per le linee seriali solo alcuni valori di velocità sono validi; questi possono essere specificati direttamente (la *glibc* prevede che i valori siano indicati in bit per secondo), ma in generale altre versioni di libreria possono utilizzare dei valori diversi. Per questo POSIX.1 prevede una serie di costanti che però servono solo per specificare le velocità tipiche delle linee seriali:

B0	B50	B75	B110	B134	B150	B200
B300	B600	B1200	B1800	B2400	B4800	B9600
B19200	B38400	B57600	B115200	B230400	B460800	

Un terminale può utilizzare solo alcune delle velocità possibili, le funzioni però non controllano se il valore specificato è valido, dato che non possono sapere a quale terminale le velocità saranno applicate; sarà l'esecuzione di `tcsetattr` a fallire quando si cercherà di eseguire l'impostazione.

Di norma il valore ha senso solo per i terminali seriali dove indica appunto la velocità della linea di trasmissione; se questa non corrisponde a quella del terminale quest'ultimo non potrà funzionare: quando il terminale non è seriale il valore non influisce sulla velocità di trasmissione dei dati.

In generale impostare un valore nullo (B0) sulla linea di output fa sì che il modem non asserisca più le linee di controllo, interrompendo di fatto la connessione, qualora invece si utilizzi questo valore per la linea di input l'effetto sarà quello di rendere la sua velocità identica a quella della linea di output.

Dato che in genere si imposta sempre la stessa velocità sulle linee di uscita e di ingresso è supportata anche la funzione `cfsetspeed`, una estensione di BSD (la funzione origina da 4.4BSD e richiede sia definita la macro `_BSD_SOURCE`) il cui prototipo è:

```
#include <unistd.h>
#include <termios.h>
int cfsetspeed(struct termios *termios_p, speed_t speed)
                                                    Imposta la velocità delle linee seriali.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, che avviene solo quando il valore specificato non è valido.

la funzione è identica alle due precedenti ma imposta la stessa velocità sia per la linea di ingresso che per quella di uscita.

Analogamente a quanto avviene per l'impostazione, le velocità possono essere lette da una struttura `termios` utilizzando altre due funzioni, `cfgetispeed` e `cfgetospeed`, i cui prototipi sono:

```
#include <unistd.h>
#include <termios.h>
speed_t cfgetispeed(struct termios *termios_p)
                                                    Legge la velocità delle linee seriali in ingresso.
speed_t cfgetospeed(struct termios *termios_p)
                                                    Legge la velocità delle linee seriali in uscita.
```

Le funzioni ritornano la velocità della linea, non sono previste condizioni di errore.

Anche in questo caso le due funzioni estraggono i valori della velocità della linea da una struttura, il cui indirizzo è specificato dall'argomento `termios_p` che deve essere stata letta in precedenza con `tcgetattr`.

Infine sempre da BSD è stata ripresa una funzione che consente di impostare il terminale in una modalità analoga alla cosiddetta modalità “*raw*” di System V, in cui i dati in input vengono resi disponibili un carattere alla volta, e l'eco e tutte le interpretazioni dei caratteri in entrata e uscita sono disabilitate. La funzione è `cfmakeraw` ed il suo prototipo è:

```
#include <unistd.h>
#include <termios.h>
void cfmakeraw(struct termios *termios_p)
                                                    Imposta il terminale in modalità “raw”.
```

La funzione imposta solo i valori in `termios_p`, e non sono previste condizioni di errore.

Anche in questo caso la funzione si limita a preparare i valori che poi saranno impostato con una successiva chiamata a `tcsetattr`, in sostanza la funzione è equivalente a:

```
termios_p->c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                    | INLCR | IGNCR | ICRNL | IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
termios_p->c_cflag &= ~(CSIZE | PARENB);
termios_p->c_cflag |= CS8;
```

8.2.3 La gestione della disciplina di linea.

Come illustrato dalla struttura riportata in fig. 8.3 tutti i terminali hanno un insieme di funzionalità comuni, che prevedono la presenza di code di ingresso ed uscita; in generale si fa riferimento a queste funzionalità con il nome di *disciplina di linea*. Lo standard POSIX prevede alcune funzioni che permettono di intervenire direttamente sulla gestione della disciplina di linea e sull'interazione fra i dati in ingresso ed uscita e le relative code.

In generale tutte queste funzioni vengono considerate, dal punto di vista dell'accesso al terminale, come delle funzioni di scrittura, pertanto se usate da processi in background sul loro terminale di controllo provocano l'emissione di SIGTTOU, come illustrato in sez. 8.1.3, con la stessa eccezione, già vista per `tcsetattr`, che quest'ultimo sia bloccato o ignorato dal processo chiamante.

Una prima funzione, che è efficace solo in caso di terminali seriali asincroni, e non fa niente per tutti gli altri terminali, è `tcsendbreak`; il suo prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcsendbreak(int fd, int duration)
```

Genera una condizione di *break*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori EBADF o ENOTTY nel loro significato generico.

La funzione invia un flusso di bit nulli, che genera una condizione di *break*, sul terminale associato a `fd`. Un valore nullo di `duration` implica una durata del flusso fra 0.25 e 0.5 secondi, un valore diverso da zero implica una durata pari a `duration*T` dove `T` è un valore compreso fra 0.25 e 0.5 secondi. Lo standard POSIX specifica il comportamento solo nel caso si sia impostato un valore nullo per `duration`, il comportamento negli altri casi può dipendere dall'implementazione.

Le altre funzioni previste dallo standard POSIX servono a controllare il comportamento dell'interazione fra le code associate al terminale e l'utente; la prima di queste è `tcdrain`, il cui prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcdrain(int fd)
```

Attende lo svuotamento della coda di uscita.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori EBADF o ENOTTY.

La funzione blocca il processo fino a che tutto l'output presente sulla coda di uscita non è stato trasmesso al terminale associato ad `fd`.

Una seconda funzione, `tcflush`, permette svuotare immediatamente le code di cancellando tutti i dati presenti al loro interno; il suo prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcflush(int fd, int queue)
```

Cancella i dati presenti nelle code di ingresso o di uscita.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori EBADF o ENOTTY.

La funzione agisce sul terminale associato a `fd`, l'argomento `queue` permette di specificare su quale coda (ingresso, uscita o entrambe), operare. Esso può prendere i valori riportati in tab. 8.11, nel caso si specifichi la coda di ingresso cancellerà i dati ricevuti ma non ancora letti, nel caso si specifichi la coda di uscita cancellerà i dati scritti ma non ancora trasmessi.

Valore	Significato
TCIFLUSH	Cancella i dati sulla coda di ingresso.
TCOFLUSH	Cancella i dati sulla coda di uscita.
TCIOFLUSH	Cancella i dati su entrambe le code.

Tabella 8.11: Possibili valori per l'argomento `queue` della funzione `tcflush`.

L'ultima funzione dell'interfaccia che interviene sulla disciplina di linea è `tcflow`, che viene usata per sospendere la trasmissione e la ricezione dei dati sul terminale; il suo prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcflow(int fd, int action)
```

Sospende e riavvia il flusso dei dati sul terminale.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori `EBADF` o `ENOTTY`.

La funzione permette di controllare (interrompendo e facendo riprendere) il flusso dei dati fra il terminale ed il sistema sia in ingresso che in uscita. Il comportamento della funzione è regolato dall'argomento `action`, i cui possibili valori, e relativa azione eseguita dalla funzione, sono riportati in tab. 8.12.

Valore	Azione
TCOOFF	Sospende l'output.
TCOON	Riprende un output precedentemente sospeso.
TCIOFF	Il sistema trasmette un carattere di STOP, che fa interrompere la trasmissione dei dati dal terminale.
TCION	Il sistema trasmette un carattere di START, che fa riprendere la trasmissione dei dati dal terminale.

Tabella 8.12: Possibili valori per l'argomento `action` della funzione `tcflow`.

8.2.4 Operare in *modo non canonico*

Operare con un terminale in modo canonico è relativamente semplice; basta eseguire una lettura e la funzione ritornerà quando il terminale avrà completato una linea di input. Non è detto che la linea sia letta interamente (si può aver richiesto un numero inferiore di byte) ma in ogni caso nessun dato verrà perso, e il resto della linea sarà letto alla chiamata successiva.

Inoltre in modo canonico la gestione dei dati in ingresso è di norma eseguita direttamente dal kernel, che si incarica (a seconda di quanto impostato con le funzioni viste nei paragrafi precedenti) di cancellare i caratteri, bloccare e riavviare il flusso dei dati, terminare la linea quando viene ricevuti uno dei vari caratteri di terminazione (NL, EOL, EOL2, EOF).

In modo non canonico è invece compito del programma gestire tutto quanto, i caratteri NL, EOL, EOL2, EOF, ERASE, KILL, CR, REPRINT non vengono interpretati automaticamente ed inoltre, non dividendo più l'input in linee, il sistema non ha più un limite definito su quando ritornare i dati ad un processo. Per questo motivo abbiamo visto che in `c_cc` sono previsti due caratteri speciali, MIN e TIME (specificati dagli indici `VMIN` e `VTIME` in `c_cc`) che dicono al sistema di ritornare da una `read` quando è stata letta una determinata quantità di dati o è passato un certo tempo.

Come accennato nella relativa spiegazione in tab. 8.9, TIME e MIN non sono in realtà caratteri ma valori numerici. Il comportamento del sistema per un terminale in modalità non canonica prevede quattro casi distinti:

MIN > 0, TIME > 0 In questo caso MIN stabilisce il numero minimo di caratteri desiderati e TIME un tempo di attesa, in decimi di secondo, fra un carattere e l'altro. Una `read` ritorna se vengono ricevuti almeno MIN caratteri prima della scadenza di TIME (MIN è solo un limite inferiore, se la funzione ha richiesto un numero maggiore di caratteri ne possono essere restituiti di più); se invece TIME scade vengono restituiti i byte ricevuti

fino ad allora (un carattere viene sempre letto, dato che il timer inizia a scorrere solo dopo la ricezione del primo carattere).

MIN> 0, **TIME**= 0 Una `read` ritorna solo dopo che sono stati ricevuti almeno **MIN** caratteri. Questo significa che una `read` può bloccarsi indefinitamente.

MIN= 0, **TIME**> 0 In questo caso **TIME** indica un tempo di attesa dalla chiamata di `read`, la funzione ritorna non appena viene ricevuto un carattere o scade il tempo. Si noti che è possibile che `read` ritorni con un valore nullo.

MIN= 0, **TIME**= 0 In questo caso una `read` ritorna immediatamente restituendo tutti i caratteri ricevuti. Anche in questo caso può ritornare con un valore nullo.

8.3 La gestione dei terminali virtuali

Da fare.

8.3.1 I terminali virtuali

Qui vanno spiegati i terminali virtuali, `/dev/pty` e compagnia.

8.3.2 Allocazione dei terminali virtuali

Qui vanno le cose su `openpty` e compagnia.

Capitolo 9

La gestione avanzata dei processi

In questo capitolo affronteremo gli argomenti relativi alla gestione avanzata dei processi. Inizieremo con le funzioni che attengono alla gestione avanzata della sicurezza, passando poi a quelle relative all'analisi ed al controllo dell'esecuzione, e alle funzioni per le modalità avanzate di creazione dei processi e l'uso dei cosiddetti *namespace*. Infine affronteremo le *system call* attinenti ad una serie di funzionalità specialistiche come la gestione della virgola mobile, le porte di I/O ecc.

9.1 La gestione avanzata della sicurezza

Tratteremo in questa sezione le funzionalità più avanzate relative alla gestione della sicurezza ed il controllo degli accessi all'interno dei processi, a partire dalle *capabilities* e dalle funzionalità del cosiddetto *Secure Computing*. Esamineremo inoltre le altre funzionalità relative alla sicurezza come la gestione delle chiavi crittografiche e varie estensioni e funzionalità disponibili su questo argomento.

9.1.1 La gestione delle *capabilities*

Come accennato in sez. 3.2.1 l'architettura classica della gestione dei privilegi in un sistema unix-like ha il sostanziale problema di fornire all'amministratore dei poteri troppo ampi, il che comporta che anche quando si siano predisposte delle misure di protezione per in essere in grado di difendersi dagli effetti di una eventuale compromissione del sistema (come montare un filesystem in sola lettura per impedirne modifiche, o marcare un file come immutabile) una volta che questa sia stata effettuata e si siano ottenuti i privilegi di amministratore, queste misure potranno essere comunque rimosse (nei casi elencati nella precedente nota si potrà sempre rimontare il sistema in lettura-scrittura, o togliere l'attributo di immutabilità).

Il problema consiste nel fatto che nell'architettura tradizionale di un sistema unix-like i controlli di accesso sono basati su un solo livello di separazione: per i processi normali essi sono posti in atto, mentre per i processi con i privilegi di amministratore essi non vengono neppure eseguiti. Per questo motivo non era previsto alcun modo per evitare che un processo con diritti di amministratore non potesse eseguire certe operazioni, o per cedere definitivamente alcuni privilegi da un certo momento in poi.

Per risolvere questo problema sono possibili varie soluzioni, ad esempio dai kernel della serie 2.5 è stata introdotta la struttura dei *Linux Security Modules* che han permesso di aggiungere varie forme di *Mandatory Access Control* (MAC), in cui si potessero parcellizzare

e controllare nei minimi dettagli tutti i privilegi e le modalità in cui questi possono essere usati dai programmi e trasferiti agli utenti, con la creazione di varie estensioni (come *SELinux*, *Smack*, *Tomoyo*, *AppArmor*) che consentono di superare l'architettura tradizionale dei permessi basati sul modello classico del controllo di accesso chiamato *Discretionary Access Control* (DAC).

Ma già in precedenza, a partire dai kernel della serie 2.2, era stato introdotto un meccanismo, detto *capabilities*, per consentire di suddividere i vari privilegi tradizionalmente associati all'amministratore in un insieme di *capacità* distinte. L'idea era che queste capacità potessero essere abilitate e disabilitate in maniera indipendente per ciascun processo con privilegi di amministratore, permettendo così una granularità molto più fine nella distribuzione degli stessi, che evitasse la situazione originaria di “*tutto o nulla*”.

Il meccanismo completo delle *capabilities* (l'implementazione si rifà ad una bozza di quello che doveva diventare lo standard POSIX.1e, poi abbandonato) prevede inoltre la possibilità di associare le stesse ai singoli file eseguibili, in modo da poter stabilire quali capacità possono essere utilizzate quando viene messo in esecuzione uno specifico programma; ma il supporto per questa funzionalità, chiamata *file capabilities*, è stato introdotto soltanto a partire dal kernel 2.6.24. Fino ad allora doveva essere il programma stesso ad eseguire una riduzione esplicita delle sue capacità, cosa che ha reso l'uso di questa funzionalità poco diffuso, vista la presenza di meccanismi alternativi per ottenere limitazioni delle capacità dell'amministratore a livello di sistema operativo, come *SELinux*.

Con questo supporto e con le ulteriori modifiche introdotte con il kernel 2.6.25 il meccanismo delle *capabilities* è stato totalmente rivoluzionato, rendendolo più aderente alle intenzioni originali dello standard POSIX, rimuovendo il significato che fino ad allora aveva avuto la capacità *CAP_SETPCAP*, e cambiando le modalità di funzionamento del cosiddetto *capabilities bounding set*. Ulteriori modifiche sono state apportate con il kernel 2.6.26 per consentire la rimozione non ripristinabile dei privilegi di amministratore. Questo fa sì che il significato ed il comportamento del kernel finisca per dipendere dalla versione dello stesso e dal fatto che le nuove *file capabilities* siano abilitate o meno. Per capire meglio la situazione e cosa è cambiato conviene allora spiegare con maggiori dettagli come funziona il meccanismo delle *capabilities*.

Il primo passo per frazionare i privilegi garantiti all'amministratore, supportato fin dalla introduzione iniziale del kernel 2.2, è stato quello in cui a ciascun processo sono stati associati tre distinti insiemi di *capabilities*, denominati rispettivamente *permitted*, *inheritable* ed *effective*. Questi insiemi vengono mantenuti in forma di tre diverse maschere binarie,¹ in cui ciascun bit corrisponde ad una capacità diversa.

L'utilizzo di tre distinti insiemi serve a fornire una interfaccia flessibile per l'uso delle *capabilities*, con scopi analoghi a quelli per cui sono mantenuti i diversi insiemi di identificatori di sez. 3.2.2; il loro significato, che è rimasto sostanzialmente lo stesso anche dopo le modifiche seguite alla introduzione delle *file capabilities* è il seguente:

permitted l'insieme delle *capabilities* “*permesse*”, cioè l'insieme di quelle capacità che un processo può impostare come *effettive* o come *ereditabili*. Se un processo cancella una capacità da questo insieme non potrà più riassumerla.²

¹il kernel li mantiene, come i vari identificatori di sez. 3.2.2, all'interno della *task_struct* di ciascun processo (vedi fig. 3.2), nei tre campi *cap_effective*, *cap_inheritable*, *cap_permitted* del tipo *kernel_cap_t*; questo era, fino al kernel 2.6.25 definito come intero a 32 bit per un massimo di 32 *capabilities* distinte, attualmente è stato aggiornato ad un vettore in grado di mantenerne fino a 64.

²questo nei casi ordinari, sono previste però una serie di eccezioni, dipendenti anche dal tipo di supporto, che vedremo meglio in seguito dato il notevole intreccio nella casistica.

- inheritable* l'insieme delle *capabilities* “ereditabili”, cioè di quelle che verranno trasmesse come insieme delle *permesse* ad un nuovo programma eseguito attraverso una chiamata ad *exec*.
- effective* l'insieme delle *capabilities* “effettive”, cioè di quelle che vengono effettivamente usate dal kernel quando deve eseguire il controllo di accesso per le varie operazioni compiute dal processo.

Con l'introduzione delle *file capabilities* sono stati introdotti altri tre insiemi associabili a ciascun file.³ Le *file capabilities* hanno effetto soltanto quando il file che le porta viene eseguito come programma con una *exec*, e forniscono un meccanismo che consente l'esecuzione dello stesso con maggiori privilegi; in sostanza sono una sorta di estensione del *suid* bit limitato ai privilegi di amministratore. Anche questi tre insiemi sono identificati con gli stessi nomi di quello dei processi, ma il loro significato è diverso:

- permitted* (chiamato originariamente *forced*) l'insieme delle capacità che con l'esecuzione del programma verranno aggiunte alle capacità *permesse* del processo.
- inheritable* (chiamato originariamente *allowed*) l'insieme delle capacità che con l'esecuzione del programma possono essere ereditate dal processo originario (che cioè non vengono tolte dall'*inheritable set* del processo originale all'esecuzione di *exec*).
- effective* in questo caso non si tratta di un insieme ma di un unico valore logico; se attivo all'esecuzione del programma tutte le capacità che risulterebbero *permesse* verranno pure attivate, inserendole automaticamente nelle *effettive*, se disattivato nessuna capacità verrà attivata (cioè l'*effective set* resterà vuoto).

Infine come accennato, esiste un ulteriore insieme, chiamato *capabilities bounding set*, il cui scopo è quello di costituire un limite alle capacità che possono essere attivate per un programma. Il suo funzionamento però è stato notevolmente modificato con l'introduzione delle *file capabilities* e si deve pertanto prendere in considerazione una casistica assai complessa.

Per i kernel fino al 2.6.25, o se non si attiva il supporto per le *file capabilities*, il *capabilities bounding set* è un parametro generale di sistema, il cui valore viene riportato nel file */proc/sys/kernel/cap-bound*. Il suo valore iniziale è definito in sede di compilazione del kernel, e da sempre ha previsto come default la presenza di tutte le *capabilities* eccetto *CAP_SETPCAP*. In questa situazione solo il primo processo eseguito nel sistema (quello con *pid* 1, di norma */sbin/init*) ha la possibilità di modificarlo; ogni processo eseguito successivamente, se dotato dei privilegi di amministratore, è in grado soltanto di rimuovere una delle *capabilities* già presenti dell'insieme.⁴

In questo caso l'effetto complessivo del *capabilities bounding set* è che solo le capacità in esso presenti possono essere trasmesse ad un altro programma attraverso una *exec*. Questo in sostanza significa che se un qualunque programma elimina da esso una capacità, considerato che *init* (almeno nelle versioni ordinarie) non supporta la reimpostazione del *bounding set*, questa non sarà più disponibile per nessun processo a meno di un riavvio, eliminando così in forma definitiva quella capacità per tutti, compreso l'amministratore.⁵

Con il kernel 2.6.25 e le *file capabilities* il *bounding set* è diventato una proprietà di ciascun processo, che viene propagata invariata sia attraverso una *fork* che una *exec*. In questo caso

³la realizzazione viene eseguita con l'uso di uno specifico attributo esteso, *security.capability*, la cui modifica è riservata, (come illustrato in sez. 4.5.1) ai processi dotato della capacità *CAP_SYS_ADMIN*.

⁴per essere precisi occorre la capacità *CAP_SYS_MODULE*.

⁵la qual cosa, visto il default usato per il *capabilities bounding set*, significa anche che *CAP_SETPCAP* non è stata praticamente mai usata nella sua forma originale.

il file `/proc/sys/kernel/cap-bound` non esiste e `init` non ha nessun ruolo speciale, inoltre in questo caso all'avvio il valore iniziale prevede la presenza di tutte le capacità (compresa `CAP_SETPCAP`).

Con questo nuovo meccanismo il *bounding set* continua a ricoprire un ruolo analogo al precedente nel passaggio attraverso una `exec`, come limite alle capacità che possono essere aggiunte al processo in quanto presenti nel *permitted set* del programma messo in esecuzione, in sostanza il nuovo programma eseguito potrà ricevere una capacità presente nel suo *permitted set* (quello del file) solo se questa è anche nel *bounding set* (del processo). In questo modo si possono rimuovere definitivamente certe capacità da un processo, anche qualora questo dovesse eseguire un programma privilegiato che prevede di riassegnarle.

Si tenga presente però che in questo caso il *bounding set* blocca esclusivamente le capacità indicate nel *permitted set* del programma che verrebbero attivate in caso di esecuzione, e non quelle eventualmente già presenti nell'*inheritable set* del processo (ad esempio perché presenti prima di averle rimosse dal *bounding set*). In questo caso eseguendo un programma che abbia anche lui dette capacità nel suo *inheritable set* queste verrebbero assegnate.

In questa seconda versione inoltre il *bounding set* costituisce anche un limite per le capacità che possono essere aggiunte all'*inheritable set* del processo stesso con `capset`, sempre nel senso che queste devono essere presenti nel *bounding set* oltre che nel *permitted set* del processo. Questo limite vale anche per processi con i privilegi di amministratore,⁶ per i quali invece non vale la condizione che le *capabilities* da aggiungere nell'*inheritable set* debbano essere presenti nel proprio *permitted set*.⁷

Come si può notare per fare ricorso alle *capabilities* occorre comunque farsi carico di una notevole complessità di gestione, aggravata dalla presenza di una radicale modifica del loro funzionamento con l'introduzione delle *file capabilities*. Considerato che il meccanismo originale era incompleto e decisamente problematico nel caso di programmi che non ne sapessero tener conto,⁸ ci soffermeremo solo sulla implementazione completa presente a partire dal kernel 2.6.25, tralasciando ulteriori dettagli riguardo la versione precedente.

Riassumendo le regole finora illustrate tutte le *capabilities* vengono ereditate senza modifiche attraverso una `fork` mentre, indicati con `orig_*` i valori degli insiemi del processo chiamante, con `file_*` quelli del file eseguito e con `bound_set` il *capabilities bounding set*, dopo l'invocazione di `exec` il processo otterrà dei nuovi insiemi di capacità `new_*` secondo la formula espressa dal seguente pseudo-codice C:

```
new_perm = ( orig_inh & file_inh ) | ( file_perm & bound_set ) ;
new_inh  = orig_inh ;
new_eff  = file_eff ? new_perm : 0 ;
new_bound_set = bound_set ;
```

e si noti come in particolare il *capabilities bounding set* non venga comunque modificato e resti lo stesso sia attraverso una `fork` che attraverso una `exec`.

A queste regole se ne aggiungono delle altre che servono a riprodurre il comportamento tradizionale di un sistema unix-like in tutta una serie di circostanze. La prima di queste è relativa a quello che avviene quando si esegue un file senza *capabilities*; se infatti si consi-

⁶si tratta sempre di avere la *capability* `CAP_SETPCAP`.

⁷lo scopo anche in questo caso è ottenere una rimozione definitiva della possibilità di passare una capacità rimossa dal *bounding set*.

⁸il problema di sicurezza originante da questa caratteristica venne alla ribalta con `sendmail`, in cui, riuscendo a rimuovere `CAP_SETGID` dall'*inheritable set* di un processo, si ottenne di far fallire `setuid` in maniera inaspettata per il programma (che aspettandosi sempre il successo della funzione non ne controllava lo stato di uscita) con la conseguenza di fargli fare come amministratore operazioni che altrimenti sarebbero state eseguite, senza poter apportare danni, da utente normale.

derasse questo equivalente al non averne assegnata alcuna, non essendo presenti capacità né nel *permitted set* né nell'*inheritable set* del file, nell'esecuzione di un qualunque programma l'amministratore perderebbe tutti i privilegi originali dal processo.

Per questo motivo se un programma senza *capabilities* assegnate viene eseguito da un processo con *UID* reale 0, esso verrà trattato come se tanto il *permitted set* che l'*inheritable set* fossero con tutte le *capabilities* abilitate, con l'*effective set* attivo, col risultato di fornire comunque al processo tutte le capacità presenti nel proprio *bounding set*. Lo stesso avviene quando l'eseguibile ha attivo il *suid* bit ed appartiene all'amministratore, in entrambi i casi si riesce così a riottenere il comportamento classico di un sistema unix-like.

Una seconda circostanza è quella relativa a cosa succede alle *capabilities* di un processo nelle possibili transizioni da *UID* nullo a *UID* non nullo o viceversa (corrispondenti rispettivamente a cedere o riottenere i privilegi di amministratore) che si possono effettuare con le varie funzioni viste in sez. 3.2.2. In questo caso la casistica è di nuovo alquanto complessa, considerata anche la presenza dei diversi gruppi di identificatori illustrati in tab. 3.7, si avrà allora che:

1. se si passa da *UID* effettivo nullo a non nullo l'*effective set* del processo viene totalmente azzerato, se viceversa si passa da *UID* effettivo non nullo a nullo il *permitted set* viene copiato nell'*effective set*;
2. se si passa da *file system UID* nullo a non nullo verranno cancellate dall'*effective set* del processo tutte le capacità attinenti i file, e cioè *CAP_LINUX_IMMUTABLE*, *CAP_MKNOD*, *CAP_DAC_OVERRIDE*, *CAP_DAC_READ_SEARCH*, *CAP_MAC_OVERRIDE*, *CAP_CHOWN*, *CAP_FSETID* e *CAP_FOWNER* (le prime due a partire dal kernel 2.2.30), nella transizione inversa verranno invece inserite nell'*effective set* quelle capacità della precedente lista che sono presenti nel suo *permitted set*.
3. se come risultato di una transizione riguardante gli identificativi dei gruppi *real*, *saved* ed *effective* in cui si passa da una situazione in cui uno di questi era nullo ad una in cui sono tutti non nulli,⁹ verranno azzerati completamente sia il *permitted set* che l'*effective set*.

La combinazione di tutte queste regole consente di riprodurre il comportamento ordinario di un sistema di tipo Unix tradizionale, ma può risultare problematica qualora si voglia passare ad una configurazione di sistema totalmente basata sull'applicazione delle *capabilities*; in tal caso infatti basta ad esempio eseguire un programma con *suid* bit di proprietà dell'amministratore per far riottenere ad un processo tutte le capacità presenti nel suo *bounding set*, anche se si era avuta la cura di cancellarle dal *permitted set*.

Per questo motivo a partire dal kernel 2.6.26, se le *file capabilities* sono abilitate, ad ogni processo viene stata associata una ulteriore maschera binaria, chiamata *securebits flags*, su cui sono mantenuti una serie di flag (vedi tab. 9.1) il cui valore consente di modificare queste regole speciali che si applicano ai processi con *UID* nullo. La maschera viene sempre mantenuta attraverso una *fork*, mentre attraverso una *exec* viene sempre cancellato il flag *SECURE_KEEP_CAPS*.

A ciascuno dei flag di tab. 9.1 è inoltre abbinato un corrispondente flag di blocco, identificato da una costante omonima con l'estensione *_LOCKED*, la cui attivazione è irreversibile ed ha l'effetto di rendere permanente l'impostazione corrente del corrispondente flag ordinario; in sostanza con *SECURE_KEEP_CAPS_LOCKED* si rende non più modificabile *SECURE_KEEP_CAPS*, ed analogamente avviene con *SECURE_NO_SETUID_FIXUP_LOCKED* per *SECURE_NO_SETUID_FIXUP* e con *SECURE_NOROOT_LOCKED* per *SECURE_NOROOT*.

⁹in sostanza questo è il caso di quando si chiama *setuid* per rimuovere definitivamente i privilegi di amministratore da un processo.

Flag	Descrizione
SECURE_KEEP_CAPS	Il processo non subisce la cancellazione delle sue <i>capabilities</i> quando tutti i suoi <i>UID</i> passano ad un valore non nullo (regola di compatibilità per il cambio di <i>UID</i> n. 3 del precedente elenco), sostituisce il precedente uso dell'operazione <code>PR_SET_KEEPCAPS</code> di <code>prctl</code> .
SECURE_NO_SETUID_FIXUP	Il processo non subisce le modifiche delle sue <i>capabilities</i> nel passaggio da nullo a non nullo degli <i>UID</i> dei gruppi <i>effective</i> e <i>file system</i> (regole di compatibilità per il cambio di <i>UID</i> nn. 1 e 2 del precedente elenco).
SECURE_NOROOT	Il processo non assume nessuna capacità aggiuntiva quando esegue un programma, anche se ha <i>UID</i> nullo o il programma ha il <i>suid</i> bit attivo ed appartiene all'amministratore (regola di compatibilità per l'esecuzione di programmi senza <i>capabilities</i>).

Tabella 9.1: Costanti identificative dei flag che compongono la maschera dei *securebits*.

Per l'impostazione di questi flag sono state predisposte due specifiche operazioni di `prctl` (vedi sez. 9.2.1), `PR_GET_SECUREBITS`, che consente di ottenerne il valore, e `PR_SET_SECUREBITS`, che consente di modificarne il valore; per quest'ultima sono comunque necessari i privilegi di amministratore ed in particolare la capacità `CAP_SETPCAP`. Prima dell'introduzione dei *securebits* era comunque possibile ottenere lo stesso effetto di `SECURE_KEEP_CAPS` attraverso l'uso di un'altra operazione di `prctl`, `PR_SET_KEEPCAPS`.

Oltre alla gestione dei *securebits* la nuova versione delle *file capabilities* prevede l'uso di `prctl` anche per la gestione del *capabilities bounding set*, attraverso altre due operazioni dedicate, `PR_CAPBSET_READ` per controllarne il valore e `PR_CAPBSET_DROP` per modificarlo; quest'ultima di nuovo è una operazione privilegiata che richiede la capacità `CAP_SETPCAP` e che, come indica chiaramente il nome, permette solo la rimozione di una *capability* dall'insieme; per i dettagli sull'uso di tutte queste operazioni si rimanda alla rilettura di sez. 9.2.1.

Un elenco delle *capabilities* disponibili su Linux, con una breve descrizione ed il nome delle costanti che le identificano, è riportato in tab. 9.2;¹⁰ la tabella è divisa in due parti, la prima riporta le *capabilities* previste anche nella bozza dello standard POSIX1.e, la seconda quelle specifiche di Linux. Come si può notare dalla tabella alcune *capabilities* attengono a singole funzionalità e sono molto specializzate, mentre altre hanno un campo di applicazione molto vasto, che è opportuno dettagliare maggiormente.

Prima di dettagliare il significato della capacità più generiche, conviene però dedicare un discorso a parte a `CAP_SETPCAP`, il cui significato è stato completamente cambiato con l'introduzione delle *file capabilities* nel kernel 2.6.24. In precedenza questa capacità era quella che permetteva al processo che la possedeva di impostare o rimuovere le *capabilities* presenti nel suo *permitted set* su un qualunque altro processo. In realtà questo non è mai stato l'uso inteso nelle bozze dallo standard POSIX, ed inoltre, come si è già accennato, dato che questa capacità è sempre stata assente (a meno di specifiche ricompilazioni del kernel) nel *capabilities bounding set* usato di default, essa non è neanche mai stata realmente disponibile.

Con l'introduzione *file capabilities* e il cambiamento del significato del *capabilities bounding set* la possibilità di modificare le capacità di altri processi è stata completamente rimossa, e `CAP_SETPCAP` ha acquisito quello che avrebbe dovuto essere il suo significato originario, e cioè la capacità del processo di poter inserire nel suo *inheritable set* qualunque capacità presente nel *bounding set*. Oltre a questo la disponibilità di `CAP_SETPCAP` consente ad un processo di eliminare una capacità dal proprio *bounding set* (con la conseguente impossibilità successiva di eseguire programmi con quella capacità), o di impostare i *securebits* delle *capabilities*.

¹⁰l'elenco presentato questa tabella, ripreso dalla pagina di manuale (accessibile con `man capabilities`) e dalle definizioni in `include/linux/capabilities.h`, è aggiornato al kernel 3.2.

Capacità	Descrizione
CAP_AUDIT_CONTROL	Abilitare e disabilitare il controllo dell'auditing (dal kernel 2.6.11).
CAP_AUDIT_WRITE	Scrivere dati nel giornale di auditing del kernel (dal kernel 2.6.11).
CAP_BLOCK_SUSPEND	Utilizzare funzionalità che possono bloccare la sospensione del sistema (dal kernel 3.5).
CAP_CHOWN	Cambiare proprietario e gruppo proprietario di un file (vedi sez. 4.4.4).
CAP_DAC_OVERRIDE	Evitare il controllo dei permessi di lettura, scrittura ed esecuzione dei file, (vedi sez. 4.4).
CAP_DAC_READ_SEARCH	Evitare il controllo dei permessi di lettura ed esecuzione per le directory (vedi sez. 4.4).
CAP_FOWNER	Evitare il controllo della proprietà di un file per tutte le operazioni privilegiate non coperte dalle precedenti CAP_DAC_OVERRIDE e CAP_DAC_READ_SEARCH.
CAP_FSETID	Evitare la cancellazione automatica dei bit <i>suid</i> e <i>sgid</i> quando un file per i quali sono impostati viene modificato da un processo senza questa capacità e la capacità di impostare il bit <i>sgid</i> su un file anche quando questo è relativo ad un gruppo cui non si appartiene (vedi sez. 4.4.3).
CAP_KILL	Mandare segnali a qualunque processo (vedi sez. 7.3.3).
CAP_SETFCAP	Impostare le <i>capabilities</i> di un file (dal kernel 2.6.24).
CAP_SETGID	Manipolare i group ID dei processi, sia il principale che i supplementari, (vedi sez. 3.2.3) che quelli trasmessi tramite i socket <i>unix domain</i> (vedi sez. 17.2).
CAP_SETUID	Manipolare gli user ID del processo (vedi sez. 3.2.2) e di trasmettere un user ID arbitrario nel passaggio delle credenziali coi socket <i>unix domain</i> (vedi sez. 17.2).
CAP_IPC_LOCK	Effettuare il <i>memory locking</i> con le funzioni <i>mlock</i> , <i>mlockall</i> , <i>shmctl</i> , <i>mmap</i> (vedi sez. 2.2.4 e sez. 10.4.1).
CAP_IPC_OWNER	Evitare il controllo dei permessi per le operazioni sugli oggetti di intercomunicazione fra processi (vedi sez. 11.2).
CAP_LEASE	Creare dei <i>file lease</i> (vedi sez. 10.3.2) pur non essendo proprietari del file (dal kernel 2.4).
CAP_LINUX_IMMUTABLE	Impostare sui file gli attributi <i>immutable</i> e <i>append-only</i> (vedi sez. 4.4.1) se supportati.
CAP_MAC_ADMIN	Amministrare il <i>Mandatory Access Control</i> di <i>Smack</i> (dal kernel 2.6.25).
CAP_MAC_OVERRIDE	Evitare il <i>Mandatory Access Control</i> di <i>Smack</i> (dal kernel 2.6.25).
CAP_MKNOD	Creare file di dispositivo con <i>mknod</i> (vedi sez. 4.2.5) (dal kernel 2.4).
CAP_NET_ADMIN	Eseguire alcune operazioni privilegiate sulla rete.
CAP_NET_BIND_SERVICE	Porsi in ascolto su porte riservate (vedi sez. 15.2.1).
CAP_NET_BROADCAST	Consentire l'uso di socket in <i>broadcast</i> e <i>multicast</i> .
CAP_NET_RAW	Usare socket RAW e PACKET (vedi sez. 14.1.4).
CAP_SETPCAP	Effettuare modifiche privilegiate alle <i>capabilities</i> .
CAP_SYS_ADMIN	Eseguire una serie di compiti amministrativi.
CAP_SYS_BOOT	Eseguire un riavvio del sistema (vedi sez. 6.2.3).
CAP_SYS_CHROOT	Eseguire la funzione <i>chroot</i> (vedi sez. 4.5.4).
CAP_SYS_MODULE	Caricare e rimuovere moduli del kernel.
CAP_SYS_NICE	Modificare le varie priorità dei processi (vedi sez. 3.3).
CAP_SYS_PACCT	Usare le funzioni di <i>accounting</i> dei processi (vedi sez. 6.3.4).
CAP_SYS_PTRACE	La capacità di tracciare qualunque processo con <i>ptrace</i> (vedi sez. 9.2.2).
CAP_SYS_RAWIO	Operare sulle porte di I/O con <i>ioperm</i> e <i>iopl</i> (vedi sez. 9.4.2).
CAP_SYS_RESOURCE	Superare le varie limitazioni sulle risorse.
CAP_SYS_TIME	Modificare il tempo di sistema (vedi sez. 6.4).
CAP_SYS_TTY_CONFIG	Simulare un <i>hangup</i> della console, con la funzione <i>vhangup</i> .
CAP_SYSLOG	Gestire il buffer dei messaggi del kernel, (vedi sez. 8.1.5), introdotta dal kernel 2.6.38 come capacità separata da CAP_SYS_ADMIN.
CAP_WAKE_ALARM	Usare i timer di tipo <i>CLOCK_BOOTTIME_ALARM</i> e <i>CLOCK_REALTIME_ALARM</i> , vedi sez. 7.5.2 (dal kernel 3.0).

Tabella 9.2: Le costanti che identificano le *capabilities* presenti nel kernel.

La prima fra le capacità “*ampie*” che occorre dettagliare maggiormente è `CAP_FOWNER`, che rimuove le restrizioni poste ad un processo che non ha la proprietà di un file in un vasto campo di operazioni;¹¹ queste comprendono i cambiamenti dei permessi e dei tempi del file (vedi sez. 4.4.3 e sez. 4.3.4), le impostazioni degli attributi dei file e delle ACL (vedi sez. 4.5.1 e 4.5.2), poter ignorare lo *sticky bit* nella cancellazione dei file (vedi sez. 4.4.2), la possibilità di impostare il flag di `O_NOATIME` con `open` e `fcntl` (vedi sez. 5.1.2 e sez. 5.2.5) senza restrizioni.

Una seconda capacità che copre diverse operazioni, in questo caso riguardanti la rete, è `CAP_NET_ADMIN`, che consente di impostare le opzioni privilegiate dei socket (vedi sez. 16.2.2), abilitare il *multicasting* (vedi sez. 16.2.4), eseguire la configurazione delle interfacce di rete (vedi sez. 16.3.2) ed impostare la tabella di instradamento.

Una terza *capability* con vasto campo di applicazione è `CAP_SYS_ADMIN`, che copre una serie di operazioni amministrative, come impostare le quote disco (vedi sez. 4.5.3), attivare e disattivare la *swap*, montare, rimontare e smontare filesystem (vedi sez. 4.1.4), effettuare operazioni di controllo su qualunque oggetto dell’IPC di SysV (vedi sez. 11.2), operare sugli attributi estesi dei file di classe *security* o *trusted* (vedi sez. 4.5.1), specificare un *UID* arbitrario nella trasmissione delle credenziali dei socket (vedi sez. ??), assegnare classi privilegiate (`IOPRIO_CLASS_RT` e prima del kernel 2.6.25 anche `IOPRIO_CLASS_IDLE`) per lo scheduling dell’I/O (vedi sez. 3.3.5), superare il limite di sistema sul numero massimo di file aperti,¹² effettuare operazioni privilegiate sulle chiavi mantenute dal kernel (vedi sez. ??), usare la funzione `lookup_dcookie`, usare `CLONE_NEWNS` con `unshare` e `clone`, (vedi sez. 9.3.1).

Originariamente `CAP_SYS_NICE` riguardava soltanto la capacità di aumentare le priorità di esecuzione dei processi, come la diminuzione del valore di *nice* (vedi sez. 3.3.2), l’uso delle priorità *real-time* (vedi sez. 3.3.3), o l’impostazione delle affinità di processore (vedi sez. 3.3.4); ma con l’introduzione di priorità anche riguardo le operazioni di accesso al disco, e, nel caso di sistemi NUMA, alla memoria, essa viene a coprire anche la possibilità di assegnare priorità arbitrarie nell’accesso a disco (vedi sez. 3.3.5) e nelle politiche di allocazione delle pagine di memoria ai nodi di un sistema NUMA.

Infine la *capability* `CAP_SYS_RESOURCE` attiene alla possibilità di superare i limiti imposti sulle risorse di sistema, come usare lo spazio disco riservato all’amministratore sui filesystem che lo supportano, usare la funzione `ioctl` per controllare il *journaling* sul filesystem *ext3*, non subire le quote disco, aumentare i limiti sulle risorse di un processo (vedi sez. 6.3.2) e quelle sul numero di processi, ed i limiti sulle dimensioni dei messaggi delle code del SysV IPC (vedi sez. 11.2.4).

Per la gestione delle *capabilities* il kernel mette a disposizione due funzioni che permettono rispettivamente di leggere ed impostare i valori dei tre insiemi illustrati in precedenza. Queste due funzioni di sistema sono `capget` e `capset` e costituiscono l’interfaccia di gestione basso livello; i loro rispettivi prototipi sono:

<pre>#include <sys/capability.h> int capget(cap_user_header_t hdrp, cap_user_data_t datap)</pre>	Legge le <i>capabilities</i> .
<pre>int capset(cap_user_header_t hdrp, const cap_user_data_t datap)</pre>	Imposta le <i>capabilities</i> .

¹¹vale a dire la richiesta che l’*UID* effettivo del processo (o meglio l’*UID* di filesystem, vedi sez. 3.2.2) coincida con quello del proprietario.

¹²quello indicato da `/proc/sys/fs/file-max`.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EFAULT	si è indicato un puntatore sbagliato o nullo per <code>hdrp</code> o <code>datap</code> (quest'ultimo può essere nullo solo se si usa <code>capget</code> per ottenere la versione delle <i>capabilities</i> usata dal kernel).
EINVAL	si è specificato un valore non valido per uno dei campi di <code>hdrp</code> , in particolare una versione non valida della versione delle <i>capabilities</i> .
EPERM	si è tentato di aggiungere una capacità nell'insieme delle <i>capabilities</i> permesse, o di impostare una capacità non presente nell'insieme di quelle permesse negli insieme delle effettive o ereditate, o si è cercato di impostare una <i>capability</i> di un altro processo senza avere <code>CAP_SETPCAP</code> .
ESRCH	si è fatto riferimento ad un processo inesistente.

Queste due funzioni prendono come argomenti due tipi di dati dedicati, definiti come puntatori a due strutture specifiche di Linux, illustrate in fig. 9.1. Per un certo periodo di tempo era anche indicato che per poterle utilizzare fosse necessario che la macro `_POSIX_SOURCE` risultasse non definita (ed era richiesto di inserire una istruzione `#undef _POSIX_SOURCE` prima di includere `sys/capability.h`) requisito che non risulta più presente.¹³

```
#define _LINUX_CAPABILITY_VERSION_1 0x19980330
#define _LINUX_CAPABILITY_U32S_1 1

#define _LINUX_CAPABILITY_VERSION_2 0x20071026
/* deprecated - use v3 */
#define _LINUX_CAPABILITY_U32S_2 2

#define _LINUX_CAPABILITY_VERSION_3 0x20080522
#define _LINUX_CAPABILITY_U32S_3 2

typedef struct __user_cap_header_struct {
    __u32 version;
    int pid;
} *cap_user_header_t;

typedef struct __user_cap_data_struct {
    __u32 effective;
    __u32 permitted;
    __u32 inheritable;
} *cap_user_data_t;
```

Figura 9.1: Definizione delle strutture a cui fanno riferimento i puntatori `cap_user_header_t` e `cap_user_data_t` usati per l'interfaccia di gestione di basso livello delle *capabilities*.

Si tenga presente che le strutture di fig. 9.1, come i prototipi delle due funzioni `capget` e `capset`, sono soggette ad essere modificate con il cambiamento del kernel (in particolare i tipi di dati delle strutture) ed anche se finora l'interfaccia è risultata stabile, non c'è nessuna assicurazione che questa venga mantenuta,¹⁴ Pertanto se si vogliono scrivere programmi portabili che possano essere eseguiti senza modifiche o adeguamenti su qualunque versione del kernel è opportuno utilizzare le interfacce di alto livello che vedremo più avanti.

¹³e non è chiaro neanche quanto sia mai stato davvero necessario.

¹⁴viene però garantito che le vecchie funzioni continuino a funzionare.

La struttura a cui deve puntare l'argomento `hdrp` serve ad indicare, tramite il campo `pid`, il *PID* del processo del quale si vogliono leggere o modificare le *capabilities*. Con `capset` questo, se si usano le *file capabilities*, può essere solo 0 o il *PID* del processo chiamante, che sono equivalenti. Non tratteremo, essendo comunque di uso irrilevante, il caso in cui, in mancanza di tale supporto, la funzione può essere usata per modificare le *capabilities* di altri processi, per il quale si rimanda, se interessati, alla lettura della pagina di manuale.

Il campo `version` deve essere impostato al valore della versione delle stesse usata dal kernel (quello indicato da una delle costanti `_LINUX_CAPABILITY_VERSION_n` di fig. 9.1) altrimenti le funzioni ritorneranno con un errore di `EINVAL`, restituendo nel campo stesso il valore corretto della versione in uso. La versione due è comunque deprecata e non deve essere usata, ed il kernel stamperà un avviso se lo si fa.

I valori delle *capabilities* devono essere passati come maschere binarie;¹⁵ con l'introduzione delle *capabilities* a 64 bit inoltre il puntatore `datap` non può essere più considerato come relativo ad una singola struttura, ma ad un vettore di due strutture.¹⁶

Dato che le precedenti funzioni, oltre ad essere specifiche di Linux, non garantiscono la stabilità nell'interfaccia, è sempre opportuno effettuare la gestione delle *capabilities* utilizzando le funzioni di libreria a questo dedicate. Queste funzioni, che seguono quanto previsto nelle bozze dello standard POSIX.1e, non fanno parte della *glibc* e sono fornite in una libreria a parte,¹⁷ pertanto se un programma le utilizza si dovrà indicare esplicitamente al compilatore l'uso della suddetta libreria attraverso l'opzione `-lcap`.

Le funzioni dell'interfaccia alle *capabilities* definite nelle bozze dello standard POSIX.1e prevedono l'uso di un tipo di dato opaco, `cap_t`, come puntatore ai dati mantenuti nel cosiddetto *capability state*,¹⁸ in sono memorizzati tutti i dati delle *capabilities*.

In questo modo è possibile mascherare i dettagli della gestione di basso livello, che potranno essere modificati senza dover cambiare le funzioni dell'interfaccia, che fanno riferimento soltanto ad oggetti di questo tipo. L'interfaccia pertanto non soltanto fornisce le funzioni per modificare e leggere le *capabilities*, ma anche quelle per gestire i dati attraverso i *capability state*, che presentano notevoli affinità, essendo parte di bozze dello stesso standard, con quelle già viste per le ACL.

La prima funzione dell'interfaccia è quella che permette di inizializzare un *capability state*, allocando al contempo la memoria necessaria per i relativi dati. La funzione è `cap_init` ed il suo prototipo è:

```
#include <sys/capability.h>
cap_t cap_init(void)
```

Crea ed inizializza un *capability state*.

La funzione ritorna un *capability state* in caso di successo e `NULL` per un errore, nel qual caso `errno` potrà assumere solo il valore `ENOMEM`.

La funzione restituisce il puntatore `cap_t` ad uno stato inizializzato con tutte le *capabilities* azzerate. In caso di errore (cioè quando non c'è memoria sufficiente ad allocare i dati) viene restituito `NULL` ed `errno` viene impostata a `ENOMEM`.

¹⁵e si tenga presente che i valori di tab. 9.2 non possono essere combinati direttamente, indicando il numero progressivo del bit associato alla relativa capacità.

¹⁶è questo cambio di significato che ha portato a deprecare la versione 2, che con `capget` poteva portare ad un buffer overflow per vecchie applicazioni che continuavano a considerare `datap` come puntatore ad una singola struttura.

¹⁷la libreria è `libcap2`, nel caso di Debian può essere installata con il pacchetto omonimo.

¹⁸si tratta in sostanza di un puntatore ad una struttura interna utilizzata dalle librerie, i cui campi non devono mai essere acceduti direttamente.

La memoria necessaria a mantenere i dati viene automaticamente allocata da `cap_init`, ma dovrà essere disallocata esplicitamente quando non è più necessaria utilizzando, per questo l'interfaccia fornisce una apposita funzione, `cap_free`, il cui prototipo è:

```
#include <sys/capability.h>
int cap_free(void *obj_d)
```

Disalloca la memoria allocata per i dati delle *capabilities*..

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` potrà assumere solo il valore `EINVAL`.

La funzione permette di liberare la memoria allocata dalle altre funzioni della libreria sia per un *capability state*, nel qual caso l'argomento sarà un dato di tipo `cap_t`, che per una descrizione testuale dello stesso,¹⁹ nel qual caso l'argomento sarà un dato di tipo `char *`. Per questo motivo l'argomento `obj_d` è dichiarato come `void *`, per evitare la necessità di eseguire un *cast*, ma dovrà comunque corrispondere ad un puntatore ottenuto tramite le altre funzioni della libreria, altrimenti la funzione fallirà con un errore di `EINVAL`.

Infine si può creare una copia di un *capability state* ottenuto in precedenza tramite la funzione `cap_dup`, il cui prototipo è:

```
#include <sys/capability.h>
cap_t cap_dup(cap_t cap_p)
```

Duplica un *capability state* restituendone una copia.

La funzione ritorna un *capability state* in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà i valori `ENOMEM` o `EINVAL` nel loro significato generico.

La funzione crea una copia del *capability state* posto all'indirizzo `cap_p` che si è passato come argomento, restituendo il puntatore alla copia, che conterrà gli stessi valori delle *capabilities* presenti nell'originale. La memoria necessaria viene allocata automaticamente dalla funzione. Una volta effettuata la copia i due *capability state* potranno essere modificati in maniera completamente indipendente, ed alla fine delle operazioni si dovrà disallocare anche la copia, oltre all'originale.

Una seconda classe di funzioni di servizio previste dall'interfaccia sono quelle per la gestione dei dati contenuti all'interno di un *capability state*; la prima di queste è `cap_clear`, il cui prototipo è:

```
#include <sys/capability.h>
int cap_clear(cap_t cap_p)
```

Inizializza un *capability state* cancellando tutte le *capabilities*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` potrà assumere solo il valore `EINVAL`.

La funzione si limita ad azzerare tutte le *capabilities* presenti nel *capability state* all'indirizzo `cap_p` passato come argomento, restituendo uno stato *vuoto*, analogo a quello che si ottiene nella creazione con `cap_init`.

Una variante di `cap_clear` è `cap_clear_flag` che cancella da un *capability state* tutte le *capabilities* di un certo insieme fra quelli elencati a pag. 393, il suo prototipo è:

```
#include <sys/capability.h>
int cap_clear_flag(cap_t cap_p, cap_flag_t flag)
```

Cancella delle *capabilities* da un *capability state*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` potrà assumere solo il valore `EINVAL`.

¹⁹cioè quanto ottenuto tramite la funzione `cap_to_text`.

La funzione richiede che si indichi quale degli insiemi si intende cancellare da `cap_p` con l'argomento `flag`. Questo deve essere specificato con una variabile di tipo `cap_flag_t` che può assumere esclusivamente²⁰ uno dei valori illustrati in tab. 9.3.

Valore	Significato
CAP_EFFECTIVE	Capacità dell'insieme <i>effettivo</i> .
CAP_PERMITTED	Capacità dell'insieme <i>permesso</i> .
CAP_INHERITABLE	Capacità dell'insieme <i>ereditabile</i> .

Tabella 9.3: Valori possibili per il tipo di dato `cap_flag_t` che identifica gli insiemi delle *capabilities*.

Si possono inoltre confrontare in maniera diretta due diversi *capability state* con la funzione `cap_compare`; il suo prototipo è:

```
#include <sys/capability.h>
int cap_compare(cap_t cap_a, cap_t cap_b)
```

Confronta due *capability state*.

La funzione ritorna 0 se i *capability state* sono identici ed un valore positivo se differiscono, non sono previsti errori.

La funzione esegue un confronto fra i due *capability state* passati come argomenti e ritorna in un valore intero il risultato, questo è nullo se sono identici o positivo se vi sono delle differenze. Il valore di ritorno della funzione consente inoltre di per ottenere ulteriori informazioni su quali sono gli insiemi di *capabilities* che risultano differenti. Per questo si può infatti usare la apposita macro `CAP_DIFFERS`:

```
#include <sys/capability.h>
int CAP_DIFFERS(value, flag)
```

Controlla lo stato di eventuali differenze delle *capabilities* nell'insieme `flag`.

La macro richiede che si passi nell'argomento `value` il risultato della funzione `cap_compare` e in `flag` l'indicazione (coi valori di tab. 9.3) dell'insieme che si intende controllare; restituirà un valore diverso da zero se le differenze rilevate da `cap_compare` sono presenti nell'insieme indicato.

Per la gestione dei singoli valori delle *capabilities* presenti in un *capability state* l'interfaccia prevede due funzioni specifiche, `cap_get_flag` e `cap_set_flag`, che permettono rispettivamente di leggere o impostare il valore di una capacità all'interno in uno dei tre insiemi già citati; i rispettivi prototipi sono:

```
#include <sys/capability.h>
int cap_get_flag(cap_t cap_p, cap_value_t cap, cap_flag_t flag,
                 cap_flag_value_t *value_p)
```

Legge il valore di una *capability*.

```
int cap_set_flag(cap_t cap_p, cap_flag_t flag, int ncap, cap_value_t *caps,
                 cap_flag_value_t value)
```

Imposta il valore di una *capability*.

Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` potrà assumere solo il valore `EINVAL`.

In entrambe le funzioni l'argomento `cap_p` indica il puntatore al *capability state* su cui operare, mentre l'argomento `flag` indica su quale dei tre insiemi si intende operare, sempre con i valori di tab. 9.3. La capacità che si intende controllare o impostare invece deve essere specificata attraverso una variabile di tipo `cap_value_t`, che può prendere come valore uno

²⁰si tratta in effetti di un tipo enumerato, come si può verificare dalla sua definizione che si trova in `sys/capability.h`.

qualunque di quelli riportati in tab. 9.2, in questo caso però non è possibile combinare diversi valori in una maschera binaria, una variabile di tipo `cap_value_t` può indicare una sola capacità.²¹

Infine lo stato di una capacità è descritto ad una variabile di tipo `cap_flag_value_t`, che a sua volta può assumere soltanto uno²² dei valori di tab. 9.4.

Valore	Significato
CAP_CLEAR	La capacità non è impostata.
CAP_SET	La capacità è impostata.

Tabella 9.4: Valori possibili per il tipo di dato `cap_flag_value_t` che indica lo stato di una capacità.

La funzione `cap_get_flag` legge lo stato della capacità indicata dall'argomento `cap` all'interno dell'insieme indicato dall'argomento `flag` e lo restituisce come *value result argument* nella variabile puntata dall'argomento `value_p`. Questa deve essere di tipo `cap_flag_value_t` ed assumerà uno dei valori di tab. 9.4. La funzione consente pertanto di leggere solo lo stato di una capacità alla volta.

La funzione `cap_set_flag` può invece impostare in una sola chiamata più *capabilities*, anche se solo all'interno dello stesso insieme ed allo stesso valore. Per questo motivo essa prende un vettore di valori di tipo `cap_value_t` nell'argomento `caps`, la cui dimensione viene specificata dall'argomento `ncap`. Il tipo di impostazione da eseguire (cancellazione o attivazione) per le capacità elencate in `caps` viene indicato dall'argomento `value` sempre con i valori di tab. 9.4.

Per semplificare la gestione delle *capabilities* l'interfaccia prevede che sia possibile utilizzare anche una rappresentazione testuale del contenuto di un *capability state* e fornisce le opportune funzioni di gestione;²³ la prima di queste, che consente di ottenere la rappresentazione testuale, è `cap_to_text`, il cui prototipo è:

```
#include <sys/capability.h>
char *cap_to_text(cap_t caps, ssize_t *length_p)
                                     Genera una visualizzazione testuale delle capabilities.
```

La funzione ritorna un puntatore alla stringa con la descrizione delle *capabilities* in caso di successo e NULL per un errore, nel qual caso `errno` assumerà i valori `EINVAL` o `ENOMEM` nel loro significato generico.

La funzione ritorna l'indirizzo di una stringa contenente la descrizione testuale del contenuto del *capability state* `caps` passato come argomento, e, qualora l'argomento `length_p` sia diverso da NULL, restituisce come *value result argument* nella variabile intera da questo puntata la lunghezza della stringa. La stringa restituita viene allocata automaticamente dalla funzione e pertanto dovrà essere liberata con `cap_free`.

La rappresentazione testuale, che viene usata anche dai programmi di gestione a riga di comando, prevede che lo stato venga rappresentato con una stringa di testo composta da una serie di proposizioni separate da spazi, ciascuna delle quali specifica una operazione da eseguire per creare lo stato finale. Nella rappresentazione si fa sempre conto di partire da uno stato in cui tutti gli insiemi sono vuoti e si provvede a impostarne i contenuti.

Ciascuna proposizione è nella forma di un elenco di capacità, espresso con i nomi di tab. 9.2 separati da virgole, seguito da un operatore, e dall'indicazione degli insiemi a cui l'operazione si applica. I nomi delle capacità possono essere scritti sia maiuscoli che minuscoli, viene inoltre riconosciuto il nome speciale `all` che è equivalente a scrivere la lista completa. Gli insiemi

²¹in `sys/capability.h` il tipo `cap_value_t` è definito come `int`, ma i valori validi sono soltanto quelli di tab. 9.2.

²²anche questo è un tipo enumerato.

²³entrambe erano previste dalla bozza dello standard POSIX.1e.

sono identificati dalle tre lettere iniziali: “p” per il *permitted*, “i” per l’*inheritable* ed “e” per l’*effective* che devono essere sempre minuscole, e se ne può indicare più di uno.

Gli operatori possibili sono solo tre: “+” che aggiunge le capacità elencate agli insiemi indicati, “-” che le toglie e “=” che le assegna esattamente. I primi due richiedono che sia sempre indicato sia un elenco di capacità che gli insiemi a cui esse devono applicarsi, e rispettivamente attiveranno o disattiveranno le capacità elencate nell’insieme o negli insiemi specificati, ignorando tutto il resto. I due operatori possono anche essere combinati nella stessa proposizione, per aggiungere e togliere le capacità dell’elenco da insiemi diversi.

L’assegnazione si applica invece su tutti gli insiemi allo stesso tempo, pertanto l’uso di “=” è equivalente alla cancellazione preventiva di tutte le capacità ed alla impostazione di quelle elencate negli insiemi specificati, questo significa che in genere lo si usa una sola volta all’inizio della stringa. In tal caso l’elenco delle capacità può non essere indicato e viene assunto che si stia facendo riferimento a tutte quante senza doverlo scrivere esplicitamente.

Come esempi avremo allora che un processo non privilegiato di un utente, che non ha nessuna capacità attiva, avrà una rappresentazione nella forma “=” che corrisponde al fatto che nessuna capacità viene assegnata a nessun insieme (vale la cancellazione preventiva), mentre un processo con privilegi di amministratore avrà una rappresentazione nella forma “=ep” in cui tutte le capacità vengono assegnate agli insiemi *permitted* ed *effective* (e l’*inheritable* è ignorato in quanto per le regole viste a pag. 9.1.1 le capacità verranno comunque attivate attraverso una *exec*). Infine, come esempio meno banale dei precedenti, otterremo per *init* una rappresentazione nella forma “=ep cap_setpcap-e” dato che come accennato tradizionalmente *CAP_SETPCAP* è sempre stata rimossa da detto processo.

Viceversa per ottenere un *capability state* dalla sua rappresentazione testuale si può usare la funzione *cap_from_text*, il cui prototipo è:

```
#include <sys/capability.h>
cap_t cap_from_text(const char *string)
    Crea un capability state dalla sua rappresentazione testuale.
```

La funzione ritorna un *capability state* in caso di successo e NULL per un errore, nel qual caso *errno* assumerà i valori *EINVAL* o *ENOMEM* nel loro significato generico.

La funzione restituisce il puntatore ad un *capability state* inizializzato con i valori indicati nella stringa *string* che ne contiene la rappresentazione testuale. La memoria per il *capability state* viene allocata automaticamente dalla funzione e dovrà essere liberata con *cap_free*.

Alle due funzioni citate se ne aggiungono altre due che consentono di convertire i valori delle costanti di tab. 9.2 nelle stringhe usate nelle rispettive rappresentazioni e viceversa. Le due funzioni, *cap_to_name* e *cap_from_name*, sono estensioni specifiche di Linux ed i rispettivi prototipi sono:

```
#include <sys/capability.h>
char *cap_to_name(cap_value_t cap)
    Convertire il valore numerico di una capabilities alla sua rappresentazione testuale.
int cap_from_name(const char *name, cap_value_t *cap_p)
    Convertire la rappresentazione testuale di una capabilities al suo valore numerico.
```

La funzione *cap_to_name* ritorna un puntatore ad una stringa in caso di successo e NULL per un errore, mentre *cap_to_name* ritorna 0 in caso di successo e -1 per un errore, per entrambe in caso di errore *errno* assumerà i valori *EINVAL* o *ENOMEM* nel loro significato generico.

La prima funzione restituisce la stringa (allocata automaticamente e che dovrà essere liberata con *cap_free*) che corrisponde al valore della capacità *cap*, mentre la seconda restituisce nella variabile puntata da *cap_p*, come *value result argument*, il valore della capacità rappresentata dalla stringa *name*.

Fin qui abbiamo trattato solo le funzioni di servizio relative alla manipolazione dei *capability state* come strutture di dati; l'interfaccia di gestione prevede però anche le funzioni per trattare le *capabilities* presenti nei processi. La prima di queste funzioni è `cap_get_proc` che consente la lettura delle *capabilities* del processo corrente, il suo prototipo è:

```
#include <sys/capability.h>
cap_t cap_get_proc(void)
```

Legge le *capabilities* del processo corrente.

La funzione ritorna un *capability state* in caso di successo e NULL per un errore, nel qual caso `errno` assumerà i valori EINVAL, EPERM o ENOMEM nel loro significato generico.

La funzione legge il valore delle *capabilities* associate al processo da cui viene invocata, restituendo il risultato tramite il puntatore ad un *capability state* contenente tutti i dati che provvede ad allocare autonomamente e che di nuovo occorrerà liberare con `cap_free` quando non sarà più utilizzato.

Se invece si vogliono leggere le *capabilities* di un processo specifico occorre usare la funzione `cap_get_pid`, il cui prototipo²⁴ è:

```
#include <sys/capability.h>
cap_t cap_get_pid(pid_t pid)
```

Legge le *capabilities* di un processo.

La funzione ritorna un *capability state* in caso di successo e NULL per un errore, nel qual caso `errno` assumerà i valori ESRCH o ENOMEM nel loro significato generico.

La funzione legge il valore delle *capabilities* del processo indicato con l'argomento `pid`, e restituisce il risultato tramite il puntatore ad un *capability state* contenente tutti i dati che provvede ad allocare autonomamente e che al solito deve essere disallocato con `cap_free`. Qualora il processo indicato non esista si avrà un errore di ESRCH. Gli stessi valori possono essere letti direttamente nel filesystem *proc*, nei file `/proc/<pid>/status`; ad esempio per `init` si otterrà qualcosa del tipo:

```
piccardi@hain:~/gapil$ cat /proc/1/status
...
CapInh: 0000000000000000
CapPrm: 00000000ffffff
CapEff: 00000000ffffff
...
```

Infine per impostare le *capabilities* del processo corrente (nella bozza dello standard POSIX.1e non esiste una funzione che permetta di cambiare le *capabilities* di un altro processo) si deve usare la funzione `cap_set_proc`, il cui prototipo è:

```
#include <sys/capability.h>
int cap_set_proc(cap_t cap_p)
```

Imposta le *capabilities* del processo corrente.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

EPERM si è cercato di attivare una capacità non permessa.
ed inoltre EINVAL nel suo significato generico.

La funzione modifica le *capabilities* del processo corrente secondo quanto specificato con l'argomento `cap_p`, posto che questo sia possibile nei termini spiegati in precedenza (non sarà ad esempio possibile impostare capacità non presenti nell'insieme di quelle permesse).

²⁴su alcune pagine di manuale la funzione è descritta con un prototipo sbagliato, che prevede un valore di ritorno di tipo `cap_t`, ma il valore di ritorno è intero, come si può verificare anche dalla dichiarazione della stessa in `sys/capability.h`.

In caso di successo i nuovi valori saranno effettivi al ritorno della funzione, in caso di fallimento invece lo stato delle capacità resterà invariato. Si tenga presente che *tutte* le capacità specificate tramite `cap_p` devono essere permesse; se anche una sola non lo è la funzione fallirà, e per quanto appena detto, lo stato delle *capabilities* non verrà modificato (neanche per le parti eventualmente permesse).

Oltre a queste funzioni su Linux sono presenti due ulteriori funzioni, `capgetp` e `capsetp`, che svolgono un compito analogo. Queste funzioni risalgono alla implementazione iniziale delle *capabilities* ed in particolare `capsetp` consentirebbe anche, come possibile in quel caso, di cambiare le capacità di un altro processo. Le due funzioni oggi sono deprecate e pertanto eviteremo di trattarle, per chi fosse interessato si rimanda alla lettura della loro pagina di manuale.

Come esempio di utilizzo di queste funzioni nei sorgenti allegati alla guida si è distribuito il programma `getcap.c`, che consente di leggere le *capabilities* del processo corrente²⁵ o tramite l'opzione `-p`, quelle di un processo qualunque il cui *PID* viene passato come parametro dell'opzione.

```

1  if (!pid) {
2      capab = cap_get_proc();
3      if (capab == NULL) {
4          perror("cannot_get_current_process_capabilities");
5          return 1;
6      }
7  } else {
8      capab = cap_get_pid(pid);
9      if (capab == NULL) {
10         perror("cannot_get_process_capabilities");
11         return 1;
12     }
13 }
14
15 string = cap_to_text(capab, NULL);
16 printf("Capability:_%s\n", string);
17
18 cap_free(capab);
19 cap_free(string);
20 return 0;

```

Figura 9.2: Corpo principale del programma `getcap.c`.

La sezione principale del programma è riportata in fig. 9.2, e si basa su una condizione sulla variabile `pid` che se si è usato l'opzione `-p` è impostata (nella sezione di gestione delle opzioni, che si è tralasciata) al valore del *PID* del processo di cui si vuole leggere le *capabilities* e nulla altrimenti. Nel primo caso (1-6) si utilizza (2) `cap_get_proc` per ottenere lo stato delle capacità del processo, nel secondo (7-13) si usa invece `cap_get_pid` (8) per leggere il valore delle capacità del processo indicato.

Il passo successivo è utilizzare (15) `cap_to_text` per tradurre in una stringa lo stato, e poi (16) stamparlo; infine (18-19) si libera la memoria allocata dalle precedenti funzioni con `cap_free` per poi ritornare dal ciclo principale della funzione.

²⁵vale a dire di sé stesso, quando lo si lancia, il che può sembrare inutile, ma serve a mostrarci quali sono le *capabilities* standard che ottiene un processo lanciato dalla riga di comando.

9.1.2 La gestione del *Secure Computing*.

Il *secure computing mode* è un meccanismo ideato per fornire un supporto per l'esecuzione di codice esterno non fidato e non verificabile a scopo di calcolo. L'idea era quella di disporre di una modalità di esecuzione dei programmi che permettesse di vendere la capacità di calcolo della propria macchina ad un qualche servizio di calcolo distribuito, senza comprometterne la sicurezza eseguendo codice non sotto il proprio controllo.

La prima versione del meccanismo è stata introdotta con il kernel 2.6.23,²⁶ è molto semplice, il *secure computing mode* viene attivato con `prctl` usando l'opzione `PR_SET_SECCOMP`, ed indicando `SECCOMP_MODE_STRICT` come valore per `arg2` (all'epoca unico valore possibile). Una volta abilitato in questa modalità (in seguito denominata *strict mode*) il processo o il *thread* chiamante potrà utilizzare soltanto un insieme estremamente limitato di *system call*: `read`, `write`, `_exit` e `sigreturn`; l'esecuzione di qualsiasi altra *system call* comporta l'emissione di un `SIGKILL` e conseguente terminazione immediata del processo.

Si tenga presente che in questo caso, con versioni recenti della *glibc* (il comportamento è stato introdotto con la 2.3), diventa impossibile usare anche `_exit` in *strict mode*, in quanto questa funzione viene intercettata ed al suo posto viene chiamata `exit_group` (vedi sez. 12.3.2) che non è consentita e comporta un `SIGKILL`.

Si tenga presente che, non essendo `execve` fra le funzioni permesse, per poter eseguire un programma terzo essendo in *strict mode* questo dovrà essere fornito in una forma di codice interpretabile fornito attraverso un socket o una *pipe*, creati prima di lanciare il processo che eseguirà il codice non fidato.

9.1.3 Altre funzionalità di sicurezza.

Oltre alle funzionalità specifiche esaminate nelle sezioni precedenti, il kernel supporta una varietà di ulteriori impostazioni di sicurezza, accessibili nelle maniere più varie, che abbiamo raccolto in questa sezione.

Una serie di modalità di sicurezza sono attivabili a richiesta attraverso alcune opzioni di controllo attivabili via `sysctl` o il filesystem `/proc`, un elenco delle stesse e dei loro effetti è il seguente:

`protected_hardlinks`

Un valore nullo, il default, mantiene il comportamento standard che non pone restrizioni alla creazione di *hard link*. Se il valore viene posto ad 1 vengono invece attivate una serie di restrizioni protettive, denominate *protected hardlinks*, che se non soddisfatte causano il fallimento di `link` con un errore di `EPERM`. Perché questo non avvenga almeno una delle seguenti condizioni deve essere soddisfatta:

- il chiamante deve avere privilegi amministrativi (la *capability* `CAP_FOWNER`). In caso di utilizzo dell'*user namespace* oltre a possedere `CAP_FOWNER` è necessario che l'*UID* del proprietario del file sia mappato nel *namespace*.
- il *filesystem UID* del chiamante (normalmente equivalente all'*UID* effettivo) deve corrispondere a quello del proprietario del file a cui si vuole effettuare il collegamento.
- devono essere soddisfatte tutte le seguenti condizioni:
 - il file è un file ordinario
 - il file non ha il *suid* bit attivo
 - il file non ha lo *sgid* bit attivo ed il permesso di esecuzione per il gruppo

²⁶e disponibile solo avendo abilitato il supporto nel kernel con l'opzione di configurazione `CONFIG_SECCOMP`.

- il chiamante ha i permessi di lettura e scrittura sul file

In sostanza in questo caso un utente potrà creare un collegamento diretto ad un altro file solo se ne è il proprietario o se questo è un file ordinario senza permessi speciali ed a cui ha accesso in lettura e scrittura.

Questa funzionalità fornisce una protezione generica che non inficia l'uso ordinario di link, ma rende impraticabili una serie di possibili abusi della stessa; oltre ad impedire l'uso di un *hard link* come variante in un attacco di *symlink race* (eludendo i *protected symlinks* di cui al punto successivo), evita anche che si possa lasciare un riferimento ad un eventuale programma *suid* vulnerabile, creando un collegamento diretto allo stesso.

protected_symlinks

Un valore nullo, il default, mantiene il comportamento standard che non pone restrizioni nel seguire i link simbolici. Se il valore viene posto ad 1 vengono attivate delle restrizioni protettive, denominate *protected symlinks*. Quando vengono attivate una qualunque funzione che esegua la risoluzione di un *pathname* contenente un link simbolico non conforme alle restrizioni fallirà con un errore di EACCESS. Per evitare l'errore deve essere soddisfatta una delle seguenti condizioni:

- il link non è in una directory con permessi analoghi a */tmp* (scrivibile a tutti e con lo *sticky bit* attivo);
- il link è in una directory con permessi analoghi a */tmp* ma è soddisfatta una delle condizioni seguenti:
 - il link simbolico appartiene al chiamante: il controllo viene fatto usando il *filesystem UID* (che normalmente corrisponde all'*UID* effettivo).
 - il link simbolico ha lo stesso proprietario della directory.

Questa funzionalità consente di rendere impraticabili alcuni attacchi in cui si approfitta di una differenza di tempo fra il controllo e l'uso di un file, ed in particolare quella classe di attacchi viene usualmente chiamati *symlink attack*,²⁷ di cui abbiamo parlato in sez. 4.2.6.

Un possibile esempio di questo tipo di attacco è quello contro un programma che viene eseguito per conto di un utente privilegiato (ad esempio un programma con il *suid* o lo *sgid* bit attivi) che prima controlla l'esistenza di un file e se non esiste lo crea. Se questa procedura, che è tipica della creazione di file temporanei sotto */tmp*, non viene eseguita in maniera corretta,²⁸ un attaccante ha una finestra di tempo in cui può creare prima del programma un *link simbolico* ad un file di sua scelta, compresi file di dispositivo o file a cui non avrebbe accesso, facendolo poi utilizzare al programma.

Attivando la funzionalità si rende impossibile seguire un link simbolico in una directory temporanea come */tmp*, a meno che questo non sia di proprietà del chiamante, o che questo non appartenga al proprietario della directory. Questo impedisce che i link simbolici creati da un attaccante possano essere seguiti da un programma privilegiato (perché apparterranno all'attaccante) mentre quelli creati dall'amministratore (che i genere è il proprietario di */tmp*) saranno seguiti comunque.

²⁷si tratta di un sottoinsieme di quella classe di attacchi chiamata genericamente *TOCTTOU*, acronimo appunto di *Time of check to time of use*.

²⁸ad esempio con le modalità che abbiamo trattato in sez. 4.2.6, che per quanto note da tempo continuano ad essere ignorate.

9.2 Funzioni di gestione e controllo

In questa sezione prenderemo in esame alcune specifiche *system call* dedicate al controllo dei processi sia per quanto riguarda l'impostazione di caratteristiche specialistiche, che per quanto riguarda l'analisi ed il controllo della loro esecuzione.

9.2.1 La funzione `prctl`

Benché la gestione ordinaria dei processi possa essere effettuata attraverso le funzioni che abbiamo già esaminato nei capitoli 2 e 3, esistono una serie di proprietà e caratteristiche specifiche dei processi per la cui gestione è stata predisposta una apposita *system call* che fornisce una interfaccia generica per tutte le operazioni specialistiche. La funzione di sistema è `prctl` ed il suo prototipo è:²⁹

```
#include <sys/prctl.h>
int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4,
          unsigned long arg5)
```

Esegue una operazione speciale sul processo corrente.

La funzione ritorna 0 o un valore positivo dipendente dall'operazione in caso di successo e -1 per un errore, nel qual caso `errno` assumerà valori diversi a seconda del tipo di operazione richiesta, sono possibili: `EACCESS`, `EBADF`, `EBUSY`, `EFAULT`, `EINVAL`, `ENXIO`, `EOPNOTSUPP` o `EPERM`.

La funzione ritorna in caso di successo un valore nullo o positivo, e -1 in caso di errore. Il significato degli argomenti della funzione successivi al primo, il valore di ritorno in caso di successo, il tipo di errore restituito in `errno` dipendono dall'operazione eseguita, indicata tramite il primo argomento, `option`. Questo è un valore intero che identifica l'operazione, e deve essere specificato con l'uso di una delle costanti predefinite del seguente elenco.³⁰ Tratteremo esplicitamente per ciascuna di esse il significato del il valore di ritorno in caso di successo, ma solo quando non corrisponde all'ordinario valore nullo (dato per implicito).

PR_CAPBSET_READ

Controlla la disponibilità di una delle *capabilities* (vedi sez. 9.1.1). La funzione ritorna 1 se la capacità specificata nell'argomento `arg2` (con una delle costanti di tab. 9.2) è presente nel *capabilities bounding set* del processo e zero altrimenti, se `arg2` non è un valore valido si avrà un errore di `EINVAL`. Introdotta a partire dal kernel 2.6.25.

PR_CAPBSET_DROP

Rimuove permanentemente una delle *capabilities* (vedi sez. 9.1.1) dal processo e da tutti i suoi discendenti. La funzione cancella la capacità specificata nell'argomento `arg2` con una delle costanti di tab. 9.2 dal *capabilities bounding set* del processo. L'operazione richiede i privilegi di amministratore (la capacità `CAP_SETPCAP`), altrimenti la chiamata fallirà con un errore di `EPERM`; se il valore di `arg2` non è valido o se il supporto per le *file capabilities* non è stato compilato nel kernel la chiamata fallirà con un errore di `EINVAL`. Introdotta a partire dal kernel 2.6.25.

PR_SET_DUMPABLE

Imposta il flag che determina se la terminazione di un processo a causa di un segnale per il quale è prevista la generazione di un file di *core dump* (vedi sez. 7.2.1) lo

²⁹la funzione non è standardizzata ed è specifica di Linux, anche se ne esiste una analoga in IRIX; è stata introdotta con il kernel 2.1.57.

³⁰l'elenco potrebbe non risultare aggiornato, in quanto nuove operazioni vengono aggiunte nello sviluppo del kernel.

genera effettivamente. In genere questo flag viene attivato automaticamente, ma per evitare problemi di sicurezza (la generazione di un file da parte di processi privilegiati può essere usata per sovrascriverne altri) viene cancellato quando si mette in esecuzione un programma con i bit *suid* e *sgid* attivi (vedi sez. 4.4.2) o con l'uso delle funzioni per la modifica degli *UID* dei processi (vedi sez. 3.2.2).

L'operazione è stata introdotta a partire dal kernel 2.3.20, fino al kernel 2.6.12 e per i kernel successivi al 2.6.17 era possibile usare solo un valore 0 (espresso anche come `SUID_DUMP_DISABLE`) di `arg2` per disattivare il flag ed un valore 1 (espresso anche come `SUID_DUMP_USER`) per attivarlo. Nei kernel dal 2.6.13 al 2.6.17 è stato supportato anche il valore 2, che causava la generazione di un *core dump* leggibile solo dall'amministratore, ma questa funzionalità è stata rimossa per motivi di sicurezza, in quanto consentiva ad un utente normale di creare un file di *core dump* appartenente all'amministratore in directory dove l'utente avrebbe avuto permessi di accesso. Specificando un valore diverso da 0 o 1 si ottiene un errore di `EINVAL`.

PR_GET_DUMPABLE

Ottiene come valore di ritorno della funzione lo stato corrente del flag che controlla la effettiva generazione dei *core dump*. Introdotta a partire dal kernel 2.3.20.

PR_SET_ENDIAN

Imposta la *endianness* del processo chiamante secondo il valore fornito in `arg2`. I valori possibili sono: `PR_ENDIAN_BIG` (*big endian*), `PR_ENDIAN_LITTLE` (*little endian*), e `PR_ENDIAN_PPC_LITTLE` (lo pseudo *little endian* del PowerPC). Introdotta a partire dal kernel 2.6.18, solo per architettura PowerPC.

PR_GET_ENDIAN

Ottiene il valore della *endianness* del processo chiamante, salvato sulla variabile puntata da `arg2` che deve essere passata come di tipo "`int *`". Introdotta a partire dal kernel 2.6.18, solo su PowerPC.

PR_SET_FPEMU

Imposta i bit di controllo per l'emulazione della virgola mobile su architettura ia64, secondo il valore di `arg2`, si deve passare `PR_FPEMU_NOPRINT` per emulare in maniera trasparente l'accesso alle operazioni in virgola mobile, o `PR_FPEMU_SIGFPE` per non emularle ed inviare il segnale `SIGFPE` (vedi sez. 7.2.2). Introdotta a partire dal kernel 2.4.18, solo su architettura ia64.

PR_GET_FPEMU

Ottiene il valore dei flag di controllo dell'emulazione della virgola mobile, salvato all'indirizzo puntato da `arg2`, che deve essere di tipo "`int *`". Introdotta a partire dal kernel 2.4.18, solo su architettura ia64.

PR_SET_FPEXC

Imposta la modalità delle eccezioni in virgola mobile (*floating-point exception mode*) al valore di `arg2`. I valori possibili sono:

- `PR_FP_EXC_SW_ENABLE` per usare `FPEXC` per le eccezioni,
- `PR_FP_EXC_DIV` per la divisione per zero in virgola mobile,
- `PR_FP_EXC_OVF` per gli overflow,
- `PR_FP_EXC_UND` per gli underflow,
- `PR_FP_EXC_RES` per risultati non esatti,
- `PR_FP_EXC_INV` per operazioni invalide,
- `PR_FP_EXC_DISABLED` per disabilitare le eccezioni,

- `PR_FP_EXC_NONRECOV` per usare la modalità di eccezione asincrona non recuperabile,
- `PR_FP_EXC_ASYNC` per usare la modalità di eccezione asincrona recuperabile,
- `PR_FP_EXC_PRECISE` per la modalità precisa di eccezione.³¹

Introdotta a partire dal kernel 2.4.21, solo su PowerPC.

`PR_GET_FPEXC`

Ottiene il valore della modalità delle eccezioni delle operazioni in virgola mobile, salvata all'indirizzo puntato `arg2`, che deve essere di tipo “`int *`”. Introdotta a partire dal kernel 2.4.21, solo su PowerPC.

`PR_SET_KEEPCAPS`

Consente di controllare quali *capabilities* vengono cancellate quando si esegue un cambiamento di *UID* del processo (per i dettagli si veda sez. 9.1.1, in particolare quanto illustrato a pag. 395). Un valore nullo (il default) per `arg2` comporta che vengano cancellate, il valore 1 che vengano mantenute, questo valore viene sempre cancellato attraverso una `exec`. L'uso di questo flag è stato sostituito, a partire dal kernel 2.6.26, dal flag `SECURE_KEEP_CAPS` dei *securebits* (vedi sez. 9.1.1 e l'uso di `PR_SET_SECUREBITS` più avanti) e si è impostato con essi `SECURE_KEEP_CAPS_LOCKED` si otterrà un errore di `EPERM`. Introdotta a partire dal kernel 2.2.18.

`PR_GET_KEEPCAPS`

Ottiene come valore di ritorno della funzione il valore del flag di controllo delle *capabilities* impostato con `PR_SET_KEEPCAPS`. Introdotta a partire dal kernel 2.2.18.

`PR_SET_NAME`

Imposta il nome del processo chiamante alla stringa puntata da `arg2`, che deve essere di tipo “`char *`”. Il nome può essere lungo al massimo 16 caratteri, e la stringa deve essere terminata da `NUL` se più corta. Introdotta a partire dal kernel 2.6.9.

`PR_GET_NAME`

Ottiene il nome del processo chiamante nella stringa puntata da `arg2`, che deve essere di tipo “`char *`”; si devono allocare per questo almeno 16 byte, e il nome sarà terminato da `NUL` se più corto. Introdotta a partire dal kernel 2.6.9.

`PR_SET_PDEATHSIG`

Consente di richiedere l'emissione di un segnale, che sarà ricevuto dal processo chiamante, in occorrenza della terminazione del proprio processo padre; in sostanza consente di invertire il ruolo di `SIGCHLD`. Il valore di `arg2` deve indicare il numero del segnale, o 0 per disabilitare l'emissione. Il valore viene automaticamente cancellato per un processo figlio creato con `fork`. Introdotta a partire dal kernel 2.1.57.

`PR_GET_PDEATHSIG`

Ottiene il valore dell'eventuale segnale emesso alla terminazione del padre, salvato all'indirizzo puntato `arg2`, che deve essere di tipo “`int *`”. Introdotta a partire dal kernel 2.3.15.

`PR_SET_PTRACER`

Imposta un *PID* per il “*tracer process*” usando `arg2`. Una impostazione successiva sovrascrive la precedente, ed un valore nullo cancella la disponibilità di un

³¹trattasi di gestione specialistica della gestione delle eccezioni dei calcoli in virgola mobile che, i cui dettagli al momento vanno al di là dello scopo di questo testo.

“*tracer process*”. Questa è una funzionalità fornita da “*Yama*”, uno specifico *Linux Security Modules*, e serve a consentire al processo indicato, quando le restrizioni introdotte da questo modulo sono attive, di usare `ptrace` (vedi sez. 9.2.2) sul processo chiamante, anche se quello indicato non ne è un progenitore. Il valore `PR_SET_PTRACER_ANY` consente a tutti i processi l’uso di `ptrace`. L’uso si *Yama* attiene alla gestione della sicurezza dei processi, e consente di introdurre una restrizione all’uso di `ptrace`, che è spesso sorgente di compromissioni. Si tratta di un uso specialistico che va al di là dello scopo di queste dispense, per i dettagli si consulti la documentazione su *Yama* nei sorgenti del kernel. Introdotta a partire dal kernel 3.4.

PR_SET_SECCOMP

Attiva il *secure computing mode* per il processo corrente. Introdotta a partire dal kernel 2.6.23 la funzionalità è stata ulteriormente estesa con il kernel 3.5, salvo poi diventare un sottoinsieme della *system call seccomp* a partire dal kernel 3.17. Prevede che si indichi per `arg2` il valore `SECCOMP_MODE_STRICT` (unico possibile fino al kernel 2.6.23) per selezionare il cosiddetto *strict mode* o, dal kernel 3.5, `SECCOMP_MODE_FILTER` per usare il *filter mode*. Tratteremo questa opzione nei dettagli più avanti, in sez. 9.1.2, quando affronteremo l’argomento del *Secure Computing*.

PR_GET_SECCOMP

Ottiene come valore di ritorno della funzione lo stato corrente del *secure computing mode*. Fino al kernel 3.5, quando era possibile solo lo *strict mode*, la funzione era totalmente inutile in quanto l’unico valore ottenibile era 0 in assenza di *secure computing*, dato che la chiamata di questa funzione in *strict mode* avrebbe comportato l’emissione di `SIGKILL` per il chiamante. La funzione però, a partire dal kernel 2.6.23, era stata comunque definita per eventuali estensioni future, ed infatti con l’introduzione del *filter mode* con il kernel 3.5, se essa viene inclusa nelle funzioni consentite restituisce il valore 2 quando il *secure computing mode* è attivo (se non inclusa si avrà di nuovo un `SIGKILL`).

PR_SET_SECUREBITS

Imposta i *securebits* per il processo chiamante al valore indicato da `arg2`; per i dettagli sul significato dei *securebits* si veda sez. 9.1.1, ed in particolare i valori di tab. 9.1 e la relativa trattazione. L’operazione richiede i privilegi di amministratore (la capacità `CAP_SETPCAP`), altrimenti la chiamata fallirà con un errore di `EPERM`. Introdotta a partire dal kernel 2.6.26.

PR_GET_SECUREBITS

Ottiene come valore di ritorno della funzione l’impostazione corrente per i *securebits*. Introdotta a partire dal kernel 2.6.26.

PR_SET_TIMING

Imposta il metodo di temporizzazione del processo da indicare con il valore di `arg2`, attualmente i valori possibili sono due, con `PR_TIMING_STATISTICAL` si usa il metodo statistico tradizionale, con `PR_TIMING_TIMESTAMP` il più accurato basato su dei *timestamp*, quest’ultimo però non è ancora implementato ed il suo uso comporta la restituzione di un errore di `EINVAL`. Introdotta a partire dal kernel 2.6.0-test4.

PR_GET_TIMING

Ottiene come valore di ritorno della funzione il metodo di temporizzazione del processo attualmente in uso (uno dei due valori citati per PR_SET_TIMING). Introdotta a partire dal kernel 2.6.0-test4.

PR_SET_TSC

Imposta il flag che indica se il processo chiamante può leggere il registro di processore contenente il contatore dei *timestamp* (TSC, o *Time Stamp Counter*) da indicare con il valore di `arg2`. Si deve specificare PR_TSC_ENABLE per abilitare la lettura o PR_TSC_SIGSEGV per disabilitarla con la generazione di un segnale di SIGSEGV (vedi sez. 7.2.2). La lettura viene automaticamente disabilitata se si attiva il *secure computing mode* (vedi PR_SET_SECCOMP e sez. 9.1.2). Introdotta a partire dal kernel 2.6.26, solo su x86.

PR_GET_TSC

Ottiene il valore del flag che controlla la lettura del contatore dei *timestamp*, salvato all'indirizzo puntato `arg2`, che deve essere di tipo "int *". Introdotta a partire dal kernel 2.6.26, solo su x86.

PR_SET_UNALIGN

Imposta la modalità di controllo per l'accesso a indirizzi di memoria non allineati, che in varie architetture risultano illegali, da indicare con il valore di `arg2`. Si deve specificare il valore PR_UNALIGN_NOPRINT per ignorare gli accessi non allineati, ed il valore PR_UNALIGN_SIGBUS per generare un segnale di SIGBUS (vedi sez. 7.2.2) in caso di accesso non allineato. Introdotta con diverse versioni su diverse architetture.

PR_GET_UNALIGN

Ottiene il valore della modalità di controllo per l'accesso a indirizzi di memoria non allineati, salvato all'indirizzo puntato `arg2`, che deve essere di tipo (int *). Introdotta con diverse versioni su diverse architetture.

PR_MCE_KILL

Imposta la politica di gestione degli errori dovuti a corruzione della memoria per problemi hardware. Questo tipo di errori vengono riportati dall'hardware di controllo della RAM e vengono gestiti dal kernel,³² ma devono essere opportunamente riportati ai processi che usano quella parte di RAM che presenta errori; nel caso specifico questo avviene attraverso l'emissione di un segnale di SIGBUS (vedi sez. 7.2.2).³³

Il comportamento di default prevede che per tutti i processi si applichi la politica generale di sistema definita nel file `/proc/sys/vm/memory_failure_early_kill`, ma specificando per `arg2` il valore PR_MCE_KILL_SET è possibile impostare con il contenuto di `arg3` una politica specifica del processo chiamante. Si può tornare alla politica di default del sistema utilizzando invece per `arg2` il valore PR_MCE_KILL_CLEAR. In tutti i casi, per compatibilità con eventuali estensioni future, tutti i valori degli argomenti non utilizzati devono essere esplicitamente posti a zero, pena il fallimento della chiamata con un errore di EINVAL.

In caso di impostazione di una politica specifica del processo con PR_MCE_KILL_SET i valori di `arg3` possono essere soltanto due, che corrispondono anche al valore

³²la funzionalità è disponibile solo sulle piattaforme più avanzate che hanno il supporto hardware per questo tipo di controlli.

³³in particolare viene anche impostato il valore di `si_code` in `siginfo_t` a `BUS_MCEERR_A0`; per il significato di tutto questo si faccia riferimento alla trattazione di sez. 7.4.3.

che si trova nell'impostazione generale di sistema di `memory_failure_early_kill`, con `PR_MCE_KILL_EARLY` si richiede l'emissione immediata di SIGBUS non appena viene rilevato un errore, mentre con `PR_MCE_KILL_LATE` il segnale verrà inviato solo quando il processo tenterà un accesso alla memoria corrotta. Questi due valori corrispondono rispettivamente ai valori 1 e 0 di `memory_failure_early_kill`.³⁴ Si può usare per `arg3` anche un terzo valore, `PR_MCE_KILL_DEFAULT`, che corrisponde a impostare per il processo la politica di default.³⁵ Introdotta a partire dal kernel 2.6.32.

PR_MCE_KILL_GET

Ottiene come valore di ritorno della funzione la politica di gestione degli errori dovuti a corruzione della memoria. Tutti gli argomenti non utilizzati (al momento tutti) devono essere nulli pena la ricezione di un errore di `EINVAL`. Introdotta a partire dal kernel 2.6.32.

PR_SET_CHILD_SUBREAPER

Se `arg2` è diverso da zero imposta l'attributo di *child reaper* per il processo, se nullo lo cancella. Lo stato di *child reaper* è una funzionalità, introdotta con il kernel 3.4, che consente di far svolgere al processo che ha questo attributo il ruolo di “genitore adottivo” per tutti i processi suoi “discendenti” che diventano orfani, in questo modo il processo potrà ricevere gli stati di terminazione alla loro uscita, sostituendo in questo ruolo `init` (si ricordi quanto illustrato in sez. 3.1.4). Il meccanismo è stato introdotto ad uso dei programmi di gestione dei servizi, per consentire loro di ricevere gli stati di terminazione di tutti i processi che lanciano, anche se questi eseguono una doppia `fork`; nel comportamento ordinario infatti questi verrebbero adottati da `init` ed il programma che li ha lanciati non sarebbe più in grado di riceverne lo stato di terminazione. Se un processo con lo stato di *child reaper* termina prima dei suoi discendenti, svolgerà questo ruolo il più prossimo antenato ad avere lo stato di *child reaper*,

PR_GET_CHILD_SUBREAPER

Ottiene l'impostazione relativa allo stato di *child reaper* del processo chiamante, salvata come *value result* all'indirizzo puntato da `arg2` (da indicare come di tipo `int *`). Il valore viene letto come valore logico, se diverso da 0 lo stato di *child reaper* è attivo altrimenti è disattivo. Introdotta a partire dal kernel 3.4.

9.2.2 La funzione `ptrace`

9.2.3 La funzione `kcmp`

9.3 La gestione avanzata della creazione dei processi

In questa sezione tratteremo le funzionalità avanzate relative alla creazione dei processi e del loro ambiente, sia per quanto riguarda l'utilizzo delle stesse per la creazione dei *thread* che per la gestione dei *namespace* che sono alla base dei cosiddetti *container*.

³⁴in sostanza nel primo caso viene immediatamente inviato il segnale a tutti i processi che hanno la memoria corrotta mappata all'interno del loro spazio degli indirizzi, nel secondo caso prima la pagina di memoria viene tolta dallo spazio degli indirizzi di ciascun processo, mentre il segnale viene inviato solo quei processi che tentano di accedervi.

³⁵si presume la politica di default corrente, in modo da non essere influenzati da un eventuale successivo cambiamento della stessa.

9.3.1 La *system call* clone

La funzione tradizionale con cui creare un nuovo processo in un sistema Unix-like, come illustrato in sez. 3.1.3, è `fork`, ma con l'introduzione del supporto del kernel per i *thread* (vedi cap. 12), si è avuta la necessità di una interfaccia che consentisse un maggiore controllo sulla modalità con cui vengono creati nuovi processi, che poi è stata utilizzata anche per fornire supporto per le tecnologie di virtualizzazione dei processi (i cosiddetti *container*) su cui torneremo in sez. 9.3.2.

Per questo l'interfaccia per la creazione di un nuovo processo è stata delegata ad una nuova *system call*, `sys_clone`, che consente di reimplementare anche la tradizionale `fork`. In realtà in questo caso più che di nuovi processi si può parlare della creazione di nuovi “*task*” del kernel che possono assumere la veste sia di un processo classico isolato dagli altri come quelli trattati finora, che di un *thread* in cui la memoria viene condivisa fra il processo chiamante ed il nuovo processo creato, come quelli che vedremo in sez. 12.2. Per evitare confusione fra *thread* e processi ordinari, abbiamo deciso di usare la nomenclatura *task* per indicare la unità di esecuzione generica messa a disposizione del kernel che `sys_clone` permette di creare.

La *system call* richiede soltanto due argomenti: il primo, `flags`, consente di controllare le modalità di creazione del nuovo *task*, il secondo, `child_stack`, imposta l'indirizzo dello *stack* per il nuovo *task*, e deve essere indicato quando si intende creare un *thread*. L'esecuzione del programma creato da `sys_clone` riprende, come per `fork`, da dopo l'esecuzione della stessa.

La necessità di avere uno *stack* alternativo c'è solo quando si intende creare un *thread*, in tal caso infatti il nuovo *task* vede esattamente la stessa memoria del *task* “padre”,³⁶ e nella sua esecuzione alla prima chiamata di una funzione andrebbe a scrivere sullo *stack* usato anche dal padre (si ricordi quanto visto in sez. 2.2.2 riguardo all'uso dello *stack*).

Per evitare di doversi garantire contro la evidente possibilità di *race condition* che questa situazione comporta (vedi sez. 3.4.2 per una spiegazione della problematica) è necessario che il chiamante allochi preventivamente un'area di memoria. In genere lo si fa con una `malloc` che allochi un buffer che la funzione imposterà come *stack* del nuovo processo, avendo ovviamente cura di non utilizzarlo direttamente nel processo chiamante.

In questo modo i due *task* avranno degli *stack* indipendenti e non si dovranno affrontare problematiche di *race condition*. Si tenga presente inoltre che in molte architetture di processore lo *stack* cresce verso il basso, pertanto in tal caso non si dovrà specificare per `child_stack` il puntatore restituito da `malloc`, ma un puntatore alla fine del buffer da essa allocato.

Dato che tutto ciò è necessario solo per i *thread* che condividono la memoria, la *system call*, a differenza della funzione di libreria che vedremo a breve, consente anche di passare per `child_stack` il valore `NULL`, che non imposta un nuovo *stack*. Se infatti si crea un processo, questo ottiene un suo nuovo spazio degli indirizzi (è sottinteso cioè che non si stia usando il flag `CLONE_VM` che vedremo a breve) ed in questo caso si applica la semantica del *copy on write* illustrata in sez. 3.1.3, per cui le pagine dello *stack* verranno automaticamente copiate come le altre e il nuovo processo avrà un suo *stack* totalmente indipendente da quello del padre.

Dato che l'uso principale della nuova *system call* è quello relativo alla creazione dei *thread*, la *glibc* definisce una funzione di libreria con una sintassi diversa, orientata a questo scopo, e la *system call* resta accessibile solo se invocata esplicitamente come visto in sez. 2.1.2.³⁷ La funzione di libreria si chiama semplicemente `clone` ed il suo prototipo è:

³⁶in questo caso per padre si intende semplicemente il *task* che ha eseguito `sys_clone` rispetto al *task* da essa creato, senza nessuna delle implicazioni che il concetto ha per i processi.

³⁷ed inoltre per questa *system call* non è disponibile la chiamata veloce con `vsyscall`.

```
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...
        /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ )
        Crea un nuovo processo o thread.
```

La funzione ritorna il *Thread ID* assegnato al nuovo processo in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN sono già in esecuzione troppi processi.
 EINVAL si è usata una combinazione non valida di flag o un valore nullo per `child_stack`.
 ENOMEM non c'è memoria sufficiente per creare una nuova `task_struct` o per copiare le parti del contesto del chiamante necessarie al nuovo *task*.
 EPERM non si hanno i privilegi di amministratore richiesti dai flag indicati.

La funzione prende come primo argomento `fn` il puntatore alla funzione che verrà messa in esecuzione nel nuovo processo, che può avere un unico argomento di tipo puntatore a `void`, il cui valore viene passato dal terzo argomento `arg`. Per quanto il precedente prototipo possa intimidire nella sua espressione, in realtà l'uso è molto semplice basterà definire una qualunque funzione `fn` che restituisce un intero ed ha come argomento un puntatore a `void`, e `fn(arg)` sarà eseguita in un nuovo processo.

Il nuovo processo resterà in esecuzione fintanto che la funzione `fn` non ritorna, o esegue `exit` o viene terminata da un segnale. Il valore di ritorno della funzione (o quello specificato con `exit`) verrà utilizzato come stato di uscita della funzione. I tre argomenti `ptid`, `tls` e `ctid` sono opzionali e sono presenti solo a partire dal kernel 2.6 e sono stati aggiunti come supporto per le funzioni di gestione dei *thread* (la *Native Thread Posix Library*, vedi sez. 12.2.2) nella *glibc*, essi vengono utilizzati soltanto se si sono specificati rispettivamente i flag `CLONE_PARENT_SETTID`, `CLONE_SETTLS` e `CLONE_CHILD_SETTID`.

La funzione ritorna un l'identificatore del nuovo *task*, denominato *Thread ID* (da qui in avanti *TID*) il cui significato è analogo al *PID* dei normali processi e che a questo corrisponde qualora si crei un processo ordinario e non un *thread*.

Il comportamento di `clone`, che si riflette sulle caratteristiche del nuovo processo da essa creato, è controllato principalmente dall'argomento `flags`, che deve essere specificato come maschera binaria, ottenuta con un OR aritmetico di una delle costanti del seguente elenco, che illustra quelle attualmente disponibili:³⁸

CLONE_CHILD_CLEARTID

cancella il valore del *thread ID* posto all'indirizzo dato dall'argomento `ctid`, eseguendo un riattivazione del *futex* (vedi sez. ??) a quell'indirizzo. Questo flag viene utilizzato dalla librerie di gestione dei *thread* ed è presente dal kernel 2.5.49.

CLONE_CHILD_SETTID

scrive il *TID* del *thread* figlio all'indirizzo dato dall'argomento `ctid`. Questo flag viene utilizzato dalla librerie di gestione dei *thread* ed è presente dal kernel 2.5.49.

CLONE_FILES

se impostato il nuovo processo condividerà con il padre la *file descriptor table* (vedi sez. 5.1.1), questo significa che ogni *file descriptor* aperto da un processo verrà visto anche dall'altro e che ogni chiusura o cambiamento dei *file descriptor flags* di un *file descriptor* verrà per entrambi.

Se non viene impostato il processo figlio eredita una copia della *file descriptor table* del padre e vale la semantica classica della gestione dei *file descriptor*, che

³⁸si fa riferimento al momento della stesura di questa sezione, cioè con il kernel 3.2.

costituisce il comportamento ordinario di un sistema unix-like e che illustreremo in dettaglio in sez. 5.2.1.

CLONE_FS se questo flag viene impostato il nuovo processo condividerà con il padre le informazioni relative all'albero dei file, ed in particolare avrà la stessa radice (vedi sez. 4.5.4), la stessa directory di lavoro (vedi sez. 4.2.4) e la stessa *umask* (sez. 4.4.3). Una modifica di una qualunque di queste caratteristiche in un processo, avrà effetto anche sull'altro. Se assente il nuovo processo riceverà una copia delle precedenti informazioni, che saranno così indipendenti per i due processi, come avviene nel comportamento ordinario di un sistema unix-like.

CLONE_IO se questo flag viene impostato il nuovo processo condividerà con il padre il contesto dell'I/O, altrimenti, come avviene nel comportamento ordinario con una *fork* otterrà un suo contesto dell'I/O.

Il contesto dell'I/O viene usato dagli *scheduler* di I/O (visti in sez. 3.3.5) e se questo è lo stesso per diversi processi questi vengono trattati come se fossero lo stesso, condividendo il tempo per l'accesso al disco, e possono interscambiarsi nell'accesso a disco. L'uso di questo flag consente, quando più *thread* eseguono dell'I/O per conto dello stesso processo (ad esempio con le funzioni di I/O asincrono di sez. 10.3.3), migliori prestazioni.

CLONE_PARENT

CLONE_PARENT_SETTID

CLONE_PID

CLONE_PTRACE

se questo flag viene impostato ed il processo chiamante viene tracciato (vedi sez. 9.2.2) anche il figlio viene tracciato.

CLONE_SETTLS

CLONE_SIGHAND

CLONE_STOPPED

CLONE_SYSVSEM

CLONE_THREAD

CLONE_UNTRACED

se questo flag viene impostato un processo non può più forzare **CLONE_PTRACE** su questo processo.

CLONE_VFORK

se questo flag viene impostato il chiamante viene fermato fintato che il figlio appena creato non rilascia la sua memoria virtuale con una chiamata a *exec* o *exit*, viene quindi replicato il comportamento di *vfork*.

CLONE_VM se questo flag viene impostato il nuovo processo condividerà con il padre la stessa memoria virtuale, e le scritture in memoria fatte da uno qualunque dei processi saranno visibili dall'altro, così come ogni mappatura in memoria (vedi sez. 10.4.1). Se non viene impostato il processo figlio otterrà una copia dello spazio degli indirizzi e si otterrà il comportamento ordinario di un processo di un sistema unix-like creato con la funzione `fork`.

9.3.2 La gestione dei *namespace*

Come accennato all'inizio di sez. 9.3.1 oltre al controllo delle caratteristiche dei processi usate per la creazione dei *thread*, l'uso di `clone` consente, ad uso delle nuove funzionalità di virtualizzazione dei processi, di creare nuovi “*namespace*” per una serie di proprietà generali (come l'elenco dei *PID*, l'albero dei file, i *mount point*, la rete, il sistema di IPC, ecc.).

L'uso dei “*namespace*” consente creare gruppi di processi che vedono le suddette proprietà in maniera indipendente fra loro. I processi di ciascun gruppo vengono così eseguiti come in una sorta di spazio separato da quello degli altri gruppi, che costituisce poi quello che viene chiamato un *container*.

9.4 Funzionalità avanzate e specialistiche

9.4.1 La gestione delle operazioni in virgola mobile

Da fare.

9.4.2 L'accesso alle porte di I/O

Da fare

Capitolo 10

La gestione avanzata dei file

In questo capitolo affronteremo le tematiche relative alla gestione avanzata dei file. Inizieremo con la trattazione delle problematiche del *file locking* e poi prenderemo in esame le varie funzionalità avanzate che permettono una gestione più sofisticata dell'I/O su file, a partire da quelle che consentono di gestire l'accesso contemporaneo a più file esaminando le varie modalità alternative di gestire l'I/O per concludere con la gestione dei file mappati in memoria e le altre funzioni avanzate che consentono un controllo più dettagliato delle modalità di I/O.

10.1 Il *file locking*

In sez. 5.2.1 abbiamo preso in esame le modalità in cui un sistema unix-like gestisce l'accesso concorrente ai file da parte di processi diversi. In quell'occasione si è visto come, con l'eccezione dei file aperti in *append mode*, quando più processi scrivono contemporaneamente sullo stesso file non è possibile determinare la sequenza in cui essi opereranno.

Questo causa la possibilità di una *race condition*; in generale le situazioni più comuni sono due: l'interazione fra un processo che scrive e altri che leggono, in cui questi ultimi possono leggere informazioni scritte solo in maniera parziale o incompleta; o quella in cui diversi processi scrivono, mescolando in maniera imprevedibile il loro output sul file.

In tutti questi casi il *file locking* è la tecnica che permette di evitare le *race condition*, attraverso una serie di funzioni che permettono di bloccare l'accesso al file da parte di altri processi, così da evitare le sovrapposizioni, e garantire la atomicità delle operazioni di lettura o scrittura.

10.1.1 L'*advisory locking*

La prima modalità di *file locking* che è stata implementata nei sistemi unix-like è quella che viene usualmente chiamata *advisory locking*,¹ in quanto sono i singoli processi, e non il sistema, che si incaricano di asserire e verificare se esistono delle condizioni di blocco per l'accesso ai file.

¹Stevens in [?] fa riferimento a questo argomento come al *record locking*, dizione utilizzata anche dal manuale della *glibc*; nelle pagine di manuale si parla di *discretionary file lock* per *fcntl* e di *advisory locking* per *flock*, mentre questo nome viene usato da Stevens per riferirsi al *file locking* POSIX. Dato che la dizione *record locking* è quantomeno ambigua, in quanto in un sistema Unix non esiste niente che possa fare riferimento al concetto di *record*, alla fine si è scelto di mantenere il nome *advisory locking*.

Questo significa che le funzioni `read` o `write` vengono eseguite comunque e non risentono affatto della presenza di un eventuale *lock*; pertanto è sempre compito dei vari processi che intendono usare il *file locking*, controllare esplicitamente lo stato dei file condivisi prima di accedervi, utilizzando le relative funzioni.

In generale si distinguono due tipologie di *file lock*;² la prima è il cosiddetto *shared lock*, detto anche *read lock* in quanto serve a bloccare l'accesso in scrittura su un file affinché il suo contenuto non venga modificato mentre lo si legge. Si parla appunto di *blocco condiviso* in quanto più processi possono richiedere contemporaneamente uno *shared lock* su un file per proteggere il loro accesso in lettura.

La seconda tipologia è il cosiddetto *exclusive lock*, detto anche *write lock* in quanto serve a bloccare l'accesso su un file (sia in lettura che in scrittura) da parte di altri processi mentre lo si sta scrivendo. Si parla di *blocco esclusivo* appunto perché un solo processo alla volta può richiedere un *exclusive lock* su un file per proteggere il suo accesso in scrittura.

In Linux sono disponibili due interfacce per utilizzare l'*advisory locking*, la prima è quella derivata da BSD, che è basata sulla funzione `flock`, la seconda è quella recepita dallo standard POSIX.1 (che è derivata dall'interfaccia usata in System V), che è basata sulla funzione `fcntl`. I *file lock* sono implementati in maniera completamente indipendente nelle due interfacce (in realtà con Linux questo avviene solo dalla serie 2.0 dei kernel) che pertanto possono coesistere senza interferenze.

Entrambe le interfacce prevedono la stessa procedura di funzionamento: si inizia sempre con il richiedere l'opportuno *file lock* (un *exclusive lock* per una scrittura, uno *shared lock* per una lettura) prima di eseguire l'accesso ad un file. Se il blocco viene acquisito il processo prosegue l'esecuzione, altrimenti (a meno di non aver richiesto un comportamento non bloccante) viene posto in stato di *sleep*. Una volta finite le operazioni sul file si deve provvedere a rimuovere il blocco.

La situazione delle varie possibilità che si possono verificare è riassunta in tab. 10.1, dove si sono riportati, a seconda delle varie tipologie di blocco già presenti su un file, il risultato che si avrebbe in corrispondenza di una ulteriore richiesta da parte di un processo di un blocco nelle due tipologie di *file lock* menzionate, con un successo o meno della richiesta.

Richiesta	Stato del file		
	Nessun <i>lock</i>	<i>Read lock</i>	<i>Write lock</i>
<i>Read lock</i>	esecuzione	esecuzione	blocco
<i>Write lock</i>	esecuzione	blocco	blocco

Tabella 10.1: Tipologie di *file locking*.

Si tenga presente infine che il controllo di accesso e la gestione dei permessi viene effettuata quando si apre un file, l'unico controllo residuo che si può avere riguardo il *file locking* è che il tipo di blocco che si vuole ottenere su un file deve essere compatibile con le modalità di apertura dello stesso (in lettura per un *read lock* e in scrittura per un *write lock*).

10.1.2 La funzione `flock`

La prima interfaccia per il *file locking*, quella derivata da BSD, permette di eseguire un blocco solo su un intero file; la funzione di sistema usata per richiedere e rimuovere un *file lock* è `flock`, ed il suo prototipo è:

²di seguito ci riferiremo sempre ai blocchi di accesso ai file con la nomenclatura inglese di *file lock*, o più brevemente con *lock*, per evitare confusioni linguistiche con il blocco di un processo (cioè la condizione in cui il processo viene posto in stato di *sleep*).

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Applica o rimuove un *file lock*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINTR la funzione è stata interrotta da un segnale nell'attesa dell'acquisizione di un *file lock*.

EINVAL si è specificato un valore non valido per `operation`.

ENOLCK il kernel non ha memoria sufficiente per gestire il *file lock*.

EWOLDBLOCK il file ha già un blocco attivo, e si è specificato LOCK_NB.

ed inoltre EBADF nel suo significato generico.

La funzione può essere usata per acquisire o rilasciare un *file lock* a seconda di quanto specificato tramite il valore dell'argomento `operation`; questo viene interpretato come maschera binaria, e deve essere passato costruendo il valore con un OR aritmetico delle costanti riportate in tab. 10.2.

Valore	Significato
LOCK_SH	Richiede uno <i>shared lock</i> sul file.
LOCK_EX	Richiede un <i>esclusive lock</i> sul file.
LOCK_UN	Rilascia il <i>file lock</i> .
LOCK_NB	Impedisce che la funzione si blocchi nella richiesta di un <i>file lock</i> .

Tabella 10.2: Valori dell'argomento `operation` di `flock`.

I primi due valori, LOCK_SH e LOCK_EX permettono di richiedere un *file lock* rispettivamente condiviso o esclusivo, ed ovviamente non possono essere usati insieme. Se con essi si specifica anche LOCK_NB la funzione non si bloccherà qualora il *file lock* non possa essere acquisito, ma ritornerà subito con un errore di EWOLDBLOCK. Per rilasciare un *file lock* si dovrà invece usare direttamente LOCK_UN.

Si tenga presente che non esiste una modalità per eseguire atomicamente un cambiamento del tipo di blocco (da *shared lock* a *esclusive lock*), il blocco deve essere prima rilasciato e poi richiesto, ed è sempre possibile che nel frattempo abbia successo un'altra richiesta pendente, facendo fallire la riacquisizione.

Si tenga presente infine che `flock` non è supportata per i file mantenuti su NFS, in questo caso, se si ha la necessità di utilizzare il *file locking*, occorre usare l'interfaccia del *file locking* POSIX basata su `fcntl` che è in grado di funzionare anche attraverso NFS, a condizione ovviamente che sia il client che il server supportino questa funzionalità.

La semantica del *file locking* di BSD inoltre è diversa da quella del *file locking* POSIX, in particolare per quanto riguarda il comportamento dei *file lock* nei confronti delle due funzioni `dup` e `fork`. Per capire queste differenze occorre descrivere con maggiore dettaglio come viene realizzato dal kernel il *file locking* per entrambe le interfacce.

In fig. 10.1 si è riportato uno schema essenziale dell'implementazione del *file locking* in stile BSD su Linux. Il punto fondamentale da capire è che un *file lock*, qualunque sia l'interfaccia che si usa, anche se richiesto attraverso un file descriptor, agisce sempre su di un file; perciò le informazioni relative agli eventuali *file lock* sono mantenute dal kernel a livello di *inode*, dato che questo è l'unico riferimento in comune che possono avere due processi diversi che aprono lo stesso file.

In particolare, come accennato in fig. 10.1, i *file lock* sono mantenuti in una *linked list* di strutture `file_lock`. La lista è referenziata dall'indirizzo di partenza mantenuto dal

campo `i_flock` della struttura `inode` (per le definizioni esatte si faccia riferimento al file `include/linux/fs.h` nei sorgenti del kernel). Un bit del campo `fl_flags` di specifica se si tratta di un lock in semantica BSD (`FL_FLOCK`) o POSIX (`FL_POSIX`) o un *file lease* (`FL_LEASE`, vedi sez. 10.3.2).

Figura 10.1: Schema dell'architettura del *file locking*, nel caso particolare del suo utilizzo da parte dalla funzione `flock`.

La richiesta di un *file lock* prevede una scansione della lista per determinare se l'acquisizione è possibile, ed in caso positivo l'aggiunta di un nuovo elemento (cioè l'aggiunta di una nuova struttura `file_lock`). Nel caso dei blocchi creati con `flock` la semantica della funzione prevede che sia `dup` che `fork` non creino ulteriori istanze di un *file lock* quanto piuttosto degli ulteriori riferimenti allo stesso. Questo viene realizzato dal kernel secondo lo schema di fig. 10.1, associando ad ogni nuovo *file lock* un puntatore alla voce nella *file table* da cui si è richiesto il blocco, che così ne identifica il titolare. Il puntatore è mantenuto nel campo `fl_file` di `file_lock`, e viene utilizzato solo per i *file lock* creati con la semantica BSD.

Questa struttura prevede che, quando si richiede la rimozione di un *file lock*, il kernel acconsenta solo se la richiesta proviene da un file descriptor che fa riferimento ad una voce nella *file table* corrispondente a quella registrata nel blocco. Allora se ricordiamo quanto visto in sez. 5.2.2 e sez. 5.2.1, e cioè che i file descriptor duplicati e quelli ereditati in un processo figlio puntano sempre alla stessa voce nella *file table*, si può capire immediatamente quali sono le conseguenze nei confronti delle funzioni `dup` e `fork`.

Sarà così possibile rimuovere un *file lock* attraverso uno qualunque dei file descriptor che fanno riferimento alla stessa voce nella *file table*, anche se questo è diverso da quello con cui lo si è creato,³ o se si esegue la rimozione in un processo figlio. Inoltre una volta tolto un *file lock* su un file, la rimozione avrà effetto su tutti i file descriptor che condividono la stessa voce nella *file table*, e quindi, nel caso di file descriptor ereditati attraverso una `fork`, anche per processi diversi.

Infine, per evitare che la terminazione imprevista di un processo lasci attivi dei *file lock*, quando un file viene chiuso il kernel provvede anche a rimuovere tutti i blocchi ad esso associati. Anche in questo caso occorre tenere presente cosa succede quando si hanno file descriptor duplicati; in tal caso infatti il file non verrà effettivamente chiuso (ed il blocco rimosso) fin tanto che non viene rilasciata la relativa voce nella *file table*; e questo avverrà solo quando tutti i file descriptor che fanno riferimento alla stessa voce sono stati chiusi. Quindi, nel caso ci siano duplicati o processi figli che mantengono ancora aperto un file descriptor, il *file lock* non viene rilasciato.

10.1.3 Il file locking POSIX

La seconda interfaccia per l'*advisory locking* disponibile in Linux è quella standardizzata da POSIX, basata sulla funzione di sistema `fcntl`. Abbiamo già trattato questa funzione nelle sue molteplici possibilità di utilizzo in sez. 5.2.5. Quando la si impiega per il *file locking* essa viene usata solo secondo il seguente prototipo:

³attenzione, questo non vale se il file descriptor fa riferimento allo stesso file, ma attraverso una voce diversa della *file table*, come accade tutte le volte che si apre più volte lo stesso file.

#include <fcntl.h> int fcntl(int fd, int cmd, struct flock *lock)	Applica o rimuove un <i>file lock</i> .
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
EACCES	l'operazione è proibita per la presenza di <i>file lock</i> da parte di altri processi.
EDEADLK	si è richiesto un <i>lock</i> su una regione bloccata da un altro processo che è a sua volta in attesa dello sblocco di un <i>lock</i> mantenuto dal processo corrente; si avrebbe pertanto un <i>deadlock</i> . Non è garantito che il sistema riconosca sempre questa situazione.
EINTR	la funzione è stata interrotta da un segnale prima di poter acquisire un <i>file lock</i> .
ENOLCK	il sistema non ha le risorse per il blocco: ci sono troppi segmenti di <i>lock</i> aperti, si è esaurita la tabella dei <i>file lock</i> , o il protocollo per il blocco remoto è fallito.
ed inoltre EBADF, EFAULT nel loro significato generico.	

Al contrario di quanto avviene con l'interfaccia basata su `flock` con `fcntl` è possibile bloccare anche delle singole sezioni di un file, fino al singolo byte. Inoltre la funzione permette di ottenere alcune informazioni relative agli eventuali blocchi preesistenti. Per poter fare tutto questo la funzione utilizza come terzo argomento una apposita struttura `flock` (la cui definizione è riportata in fig. 10.2) nella quale inserire tutti i dati relativi ad un determinato blocco. Si tenga presente poi che un *file lock* fa sempre riferimento ad una regione, per cui si potrà avere un conflitto anche se c'è soltanto una sovrapposizione parziale con un'altra regione bloccata.

```

struct flock {
    short int l_type;      /* Type of lock: F_RDLCK, F_WRLCK, or F_UNLCK. */
    short int l_whence;    /* Where 'l_start' is relative to (like 'lseek').*/
    off_t l_start;         /* Offset where the lock begins. */
    off_t l_len;           /* Size of the locked area; zero means until EOF.*/
    pid_t l_pid;           /* Process holding the lock. */
};

```

Figura 10.2: La struttura `flock`, usata da `fcntl` per il *file locking*.

I primi tre campi della struttura, `l_whence`, `l_start` e `l_len`, servono a specificare la sezione del file a cui fa riferimento il blocco: `l_start` specifica il byte di partenza, `l_len` la lunghezza della sezione e infine `l_whence` imposta il riferimento da cui contare `l_start`. Il valore di `l_whence` segue la stessa semantica dell'omonimo argomento di `lseek`, coi tre possibili valori `SEEK_SET`, `SEEK_CUR` e `SEEK_END`, (si vedano le relative descrizioni in tab. 5.5).

Si tenga presente che un *file lock* può essere richiesto anche per una regione al di là della corrente fine del file, così che una eventuale estensione dello stesso resti coperta dal blocco. Inoltre se si specifica un valore nullo per `l_len` il blocco si considera esteso fino alla dimensione massima del file; in questo modo è possibile bloccare una qualunque regione a partire da un certo punto fino alla fine del file, coprendo automaticamente quanto eventualmente aggiunto in coda allo stesso.

Lo standard POSIX non richiede che `l_len` sia positivo, ed a partire dal kernel 2.4.21 è possibile anche indicare valori di `l_len` negativi, in tal caso l'intervallo coperto va da `l_start+l_len` a `l_start-1`, mentre per un valore positivo l'intervallo va da `l_start` a `l_start+l_len-1`. Si può però usare un valore negativo soltanto se l'inizio della regione indicata non cade prima dell'inizio del file, mentre come accennato con un valore positivo si può anche indicare una regione che eccede la dimensione corrente del file.

Il tipo di *file lock* richiesto viene specificato dal campo `l_type`, esso può assumere i tre valori definiti dalle costanti riportate in tab. 10.3, che permettono di richiedere rispettivamente uno *shared lock*, un *exclusive lock*, e la rimozione di un blocco precedentemente acquisito. Infine il campo `l_pid` viene usato solo in caso di lettura, quando si chiama `fcntl` con `F_GETLK`, e riporta il *PID* del processo che detiene il *file lock*.

Valore	Significato
<code>F_RDLCK</code>	Richiede un blocco condiviso (<i>read lock</i>).
<code>F_WRLCK</code>	Richiede un blocco esclusivo (<i>write lock</i>).
<code>F_UNLCK</code>	Richiede l'eliminazione di un <i>file lock</i> .

Tabella 10.3: Valori possibili per il campo `l_type` di `flock`.

Oltre a quanto richiesto tramite i campi di `flock`, l'operazione effettivamente svolta dalla funzione è stabilita dal valore dall'argomento `cmd` che, come già riportato in sez. 5.2.5, specifica l'azione da compiere; i valori utilizzabili relativi al *file locking* sono tre:

- `F_GETLK` verifica se il *file lock* specificato dalla struttura puntata da `lock` può essere acquisito: in caso negativo sovrascrive la struttura `flock` con i valori relativi al blocco già esistente che ne blocca l'acquisizione, altrimenti si limita a impostarne il campo `l_type` con il valore `F_UNLCK`.
- `F_SETLK` se il campo `l_type` della struttura puntata da `lock` è `F_RDLCK` o `F_WRLCK` richiede il corrispondente *file lock*, se è `F_UNLCK` lo rilascia; nel caso la richiesta non possa essere soddisfatta a causa di un blocco preesistente la funzione ritorna immediatamente con un errore di `EACCES` o di `EAGAIN`.
- `F_SETLKW` è identica a `F_SETLK`, ma se la richiesta di non può essere soddisfatta per la presenza di un altro blocco, mette il processo in stato di attesa fintanto che il blocco precedente non viene rilasciato; se l'attesa viene interrotta da un segnale la funzione ritorna con un errore di `EINTR`.

Si noti che per quanto detto il comando `F_GETLK` non serve a rilevare una presenza generica di blocco su un file, perché se ne esistono altri compatibili con quello richiesto, la funzione ritorna comunque impostando `l_type` a `F_UNLCK`. Inoltre a seconda del valore di `l_type` si potrà controllare o l'esistenza di un qualunque tipo di blocco (se è `F_WRLCK`) o di *write lock* (se è `F_RDLCK`). Si consideri poi che può esserci più di un blocco che impedisce l'acquisizione di quello richiesto (basta che le regioni si sovrappongano), ma la funzione ne riporterà sempre soltanto uno, impostando `l_whence` a `SEEK_SET` ed i valori `l_start` e `l_len` per indicare quale è la regione bloccata.

Infine si tenga presente che effettuare un controllo con il comando `F_GETLK` e poi tentare l'acquisizione con `F_SETLK` non è una operazione atomica (un altro processo potrebbe acquisire un blocco fra le due chiamate) per cui si deve sempre verificare il codice di ritorno di `fcntl`⁴ quando la si invoca con `F_SETLK`, per controllare che il blocco sia stato effettivamente acquisito.

Figura 10.3: Schema di una situazione di *deadlock*.

Non operando a livello di interi file, il *file locking* POSIX introduce un'ulteriore complicazione; consideriamo la situazione illustrata in fig. 10.3, in cui il processo A blocca la regione

⁴controllare il codice di ritorno delle funzioni invocate è comunque una buona norma di programmazione, che permette di evitare un sacco di errori difficili da tracciare proprio perché non vengono rilevati.

1 e il processo B la regione 2. Supponiamo che successivamente il processo A richieda un lock sulla regione 2 che non può essere acquisito per il preesistente lock del processo 2; il processo 1 si bloccherà fintanto che il processo 2 non rilasci il blocco. Ma cosa accade se il processo 2 nel frattempo tenta a sua volta di ottenere un lock sulla regione A? Questa è una tipica situazione che porta ad un *deadlock*, dato che a quel punto anche il processo 2 si bloccherebbe, e niente potrebbe sbloccare l'altro processo. Per questo motivo il kernel si incarica di rilevare situazioni di questo tipo, ed impedirle restituendo un errore di EDEADLK alla funzione che cerca di acquisire un blocco che porterebbe ad un *deadlock*.

Per capire meglio il funzionamento del *file locking* in semantica POSIX (che differisce alquanto rispetto da quello di BSD, visto sez. 10.1.2) esaminiamo più in dettaglio come viene gestito dal kernel. Lo schema delle strutture utilizzate è riportato in fig. 10.4; come si vede esso è molto simile all'analogo di fig. 10.1. In questo caso nella figura si sono evidenziati solo i campi di *file_lock* significativi per la semantica POSIX, in particolare adesso ciascuna struttura contiene, oltre al *PID* del processo in *fl_pid*, la sezione di file che viene bloccata grazie ai campi *fl_start* e *fl_end*. La struttura è comunque la stessa, solo che in questo caso nel campo *fl_flags* è impostato il bit *FL_POSIX* ed il campo *fl_file* non viene usato. Il blocco è sempre associato all'*inode*, solo che in questo caso la titolarità non viene identificata con il riferimento ad una voce nella *file table*, ma con il valore del *PID* del processo.

Figura 10.4: Schema dell'architettura del *file locking*, nel caso particolare del suo utilizzo secondo l'interfaccia standard POSIX.

Quando si richiede un *file lock* il kernel effettua una scansione di tutti i blocchi presenti sul file⁵ per verificare se la regione richiesta non si sovrappone ad una già bloccata, in caso affermativo decide in base al tipo di blocco, in caso negativo il nuovo blocco viene comunque acquisito ed aggiunto alla lista.

Nel caso di rimozione invece questa viene effettuata controllando che il *PID* del processo richiedente corrisponda a quello contenuto nel blocco. Questa diversa modalità ha delle conseguenze precise riguardo il comportamento dei *file lock* POSIX. La prima conseguenza è che un *file lock* POSIX non viene mai ereditato attraverso una *fork*, dato che il processo figlio avrà un *PID* diverso, mentre passa indenne attraverso una *exec* in quanto il *PID* resta lo stesso. Questo comporta che, al contrario di quanto avveniva con la semantica BSD, quando un processo termina tutti i *file lock* da esso detenuti vengono immediatamente rilasciati.

La seconda conseguenza è che qualunque file descriptor che faccia riferimento allo stesso file (che sia stato ottenuto con una *dup* o con una *open* in questo caso non fa differenza) può essere usato per rimuovere un blocco, dato che quello che conta è solo il *PID* del processo. Da questo deriva una ulteriore sottile differenza di comportamento: dato che alla chiusura di un file i blocchi ad esso associati vengono rimossi, nella semantica POSIX basterà chiudere un file descriptor qualunque per cancellare tutti i blocchi relativi al file cui esso faceva riferimento, anche se questi fossero stati creati usando altri file descriptor che restano aperti.

Dato che il controllo sull'accesso ai blocchi viene eseguito sulla base del *PID* del processo, possiamo anche prendere in considerazione un altro degli aspetti meno chiari di questa interfaccia e cioè cosa succede quando si richiedono dei blocchi su regioni che si sovrappongono fra loro all'interno stesso processo. Siccome il controllo, come nel caso della rimozione, si basa solo sul *PID* del processo che chiama la funzione, queste richieste avranno sempre successo. Nel

⁵scandisce cioè la *linked list* delle strutture *file_lock*, scartando automaticamente quelle per cui *fl_flags* non è *FL_POSIX*, così che le due interfacce restano ben separate.

caso della semantica BSD, essendo i lock relativi a tutto un file e non accumulandosi,⁶ la cosa non ha alcun effetto; la funzione ritorna con successo, senza che il kernel debba modificare la lista dei *file lock*.

Con i *file lock* POSIX invece si possono avere una serie di situazioni diverse: ad esempio è possibile rimuovere con una sola chiamata più *file lock* distinti (indicando in una regione che si sovrapponga completamente a quelle di questi ultimi), o rimuovere solo una parte di un blocco preesistente (indicando una regione contenuta in quella di un altro blocco), creando un buco, o coprire con un nuovo blocco altri *file lock* già ottenuti, e così via, a secondo di come si sovrappongono le regioni richieste e del tipo di operazione richiesta.

Il comportamento seguito in questo caso è che la funzione ha successo ed esegue l'operazione richiesta sulla regione indicata; è compito del kernel preoccuparsi di accorpare o dividere le voci nella lista dei *file lock* per far sì che le regioni bloccate da essa risultanti siano coerenti con quanto necessario a soddisfare l'operazione richiesta.

Per fare qualche esempio sul *file locking* si è scritto un programma che permette di bloccare una sezione di un file usando la semantica POSIX, o un intero file usando la semantica BSD; in fig. 10.5 è riportata il corpo principale del codice del programma, (il testo completo è allegato nella directory dei sorgenti, nel file `Flock.c`).

La sezione relativa alla gestione delle opzioni al solito si è omessa, come la funzione che stampa le istruzioni per l'uso del programma, essa si cura di impostare le variabili `type`, `start` e `len`; queste ultime due vengono inizializzate al valore numerico fornito rispettivamente tramite gli switch `-s` e `-l`, mentre il valore della prima viene impostato con le opzioni `-w` e `-r` si richiede rispettivamente o un *write lock* o *read lock* (i due valori sono esclusivi, la variabile assumerà quello che si è specificato per ultimo). Oltre a queste tre vengono pure impostate la variabile `bsd`, che abilita la semantica omonima quando si invoca l'opzione `-f` (il valore preimpostato è nullo, ad indicare la semantica POSIX), e la variabile `cmd` che specifica la modalità di richiesta del *file lock* (bloccante o meno), a seconda dell'opzione `-b`.

Il programma inizia col controllare (11-14) che venga passato un argomento (il file da bloccare), che sia stato scelto (15-18) il tipo di blocco, dopo di che apre (19) il file, uscendo (20-23) in caso di errore. A questo punto il comportamento dipende dalla semantica scelta; nel caso sia BSD occorre reimpostare il valore di `cmd` per l'uso con `flock`; infatti il valore preimpostato fa riferimento alla semantica POSIX e vale rispettivamente `F_SETLKW` o `F_SETLK` a seconda che si sia impostato o meno la modalità bloccante.

Nel caso si sia scelta la semantica BSD (25-34) prima si controlla (27-31) il valore di `cmd` per determinare se si vuole effettuare una chiamata bloccante o meno, reimpostandone il valore opportunamente, dopo di che a seconda del tipo di blocco al valore viene aggiunta la relativa opzione, con un OR aritmetico, dato che `flock` vuole un argomento *operation* in forma di maschera binaria. Nel caso invece che si sia scelta la semantica POSIX le operazioni sono molto più immediate si prepara (36-40) la struttura per il lock, e lo si esegue (41).

In entrambi i casi dopo aver richiesto il blocco viene controllato il risultato uscendo (44-46) in caso di errore, o stampando un messaggio (47-49) in caso di successo. Infine il programma si pone in attesa (50) finché un segnale (ad esempio un C-c dato da tastiera) non lo interrompa; in questo caso il programma termina, e tutti i blocchi vengono rilasciati.

Con il programma possiamo fare varie verifiche sul funzionamento del *file locking*; cominciamo con l'eseguire una *read lock* su un file, ad esempio usando all'interno di un terminale il seguente comando:

⁶questa ultima caratteristica è vera in generale, se cioè si richiede più volte lo stesso *file lock*, o più blocchi sulla stessa sezione di file, le richieste non si cumulano e basta una sola richiesta di rilascio per cancellare il blocco.

```

1 int main(int argc, char *argv[])
2 {
3     int type = F_UNLCK; /* lock type: default to unlock (invalid) */
4     off_t start = 0;    /* start of the locked region: default to 0 */
5     off_t len = 0;      /* length of the locked region: default to 0 */
6     int fd, res, i;      /* internal variables */
7     int bsd = 0;         /* semantic type: default to POSIX */
8     int cmd = F_SETLKW; /* lock command: default to non-blocking */
9     struct flock lock;   /* file lock structure */
10    ...
11    if ((argc - optind) != 1) { /* There must be remaining parameters */
12        printf("Wrong_number_of_arguments_%d\n", argc - optind);
13        usage();
14    }
15    if (type == F_UNLCK) { /* There must be a -w or -r option set */
16        printf("You_should_set_a_read_or_a_write_lock\n");
17        usage();
18    }
19    fd = open(argv[optind], O_RDWR); /* open the file to be locked */
20    if (fd < 0) { /* on error exit */
21        perror("Wrong_filename");
22        exit(1);
23    }
24    /* do lock */
25    if (bsd) { /* if BSD locking */
26        /* rewrite cmd for suitable flock operation values */
27        if (cmd == F_SETLKW) { /* if no-blocking */
28            cmd = LOCK_NB; /* set the value for flock operation */
29        } else { /* else */
30            cmd = 0; /* default is null */
31        }
32        if (type == F_RDLCK) cmd |= LOCK_SH; /* set for shared lock */
33        if (type == F_WRLCK) cmd |= LOCK_EX; /* set for exclusive lock */
34        res = flock(fd, cmd); /* execute lock */
35    } else { /* if POSIX locking */
36        /* setting flock structure */
37        lock.l_type = type; /* set type: read or write */
38        lock.l_whence = SEEK_SET; /* start from the beginning of the file */
39        lock.l_start = start; /* set the start of the locked region */
40        lock.l_len = len; /* set the length of the locked region */
41        res = fcntl(fd, cmd, &lock); /* do lock */
42    }
43    /* check lock results */
44    if (res) { /* on error exit */
45        perror("Failed_lock");
46        exit(1);
47    } else { /* else write message */
48        printf("Lock_acquired\n");
49    }
50    pause(); /* stop the process, use a signal to exit */
51    return 0;
52 }

```

Figura 10.5: Sezione principale del codice del programma Flock.c.

```
[piccardi@gont sources]$ ./flock -r Flock.c
Lock acquired
```

il programma segnerà di aver acquisito un blocco e si bloccherà; in questo caso si è usato il *file locking* POSIX e non avendo specificato niente riguardo alla sezione che si vuole bloccare sono stati usati i valori preimpostati che bloccano tutto il file. A questo punto se proviamo ad eseguire lo stesso comando in un altro terminale, e avremo lo stesso risultato. Se invece proviamo ad eseguire un *write lock* avremo:

```
[piccardi@gont sources]$ ./flock -w Flock.c
Failed lock: Resource temporarily unavailable
```

come ci aspettiamo il programma terminerà segnalando l'indisponibilità del blocco, dato che il file è bloccato dal precedente *read lock*. Si noti che il risultato è lo stesso anche se si richiede il blocco su una sola parte del file con il comando:

```
[piccardi@gont sources]$ ./flock -w -s0 -l10 Flock.c
Failed lock: Resource temporarily unavailable
```

se invece blocchiamo una regione con:

```
[piccardi@gont sources]$ ./flock -r -s0 -l10 Flock.c
Lock acquired
```

una volta che riproviamo ad acquisire il *write lock* i risultati dipenderanno dalla regione richiesta; ad esempio nel caso in cui le due regioni si sovrappongono avremo che:

```
[piccardi@gont sources]$ ./flock -w -s5 -l15 Flock.c
Failed lock: Resource temporarily unavailable
```

ed il blocco viene rifiutato, ma se invece si richiede una regione distinta avremo che:

```
[piccardi@gont sources]$ ./flock -w -s11 -l15 Flock.c
Lock acquired
```

ed il blocco viene acquisito. Se a questo punto si prova ad eseguire un *read lock* che comprende la nuova regione bloccata in scrittura:

```
[piccardi@gont sources]$ ./flock -r -s10 -l20 Flock.c
Failed lock: Resource temporarily unavailable
```

come ci aspettiamo questo non sarà consentito.

Il programma di norma esegue il tentativo di acquisire il lock in modalità non bloccante, se però usiamo l'opzione **-b** possiamo impostare la modalità bloccante, riproviamo allora a ripetere le prove precedenti con questa opzione:

```
[piccardi@gont sources]$ ./flock -r -b -s0 -l10 Flock.c Lock acquired
```

il primo comando acquisisce subito un *read lock*, e quindi non cambia nulla, ma se proviamo adesso a richiedere un *write lock* che non potrà essere acquisito otterremo:

```
[piccardi@gont sources]$ ./flock -w -s0 -l10 Flock.c
```

il programma cioè si bloccherà nella chiamata a `fcntl`; se a questo punto rilasciamo il precedente blocco (terminando il primo comando un C-c sul terminale) potremo verificare che sull'altro terminale il blocco viene acquisito, con la comparsa di una nuova riga:

```
[piccardi@gont sources]$ ./flock -w -s0 -l10 Flock.c
Lock acquired
```

Un'altra cosa che si può controllare con il nostro programma è l'interazione fra i due tipi di blocco; se ripartiamo dal primo comando con cui si è ottenuto un blocco in lettura sull'intero file, possiamo verificare cosa succede quando si cerca di ottenere un blocco in scrittura con la semantica BSD:

```
[root@gont sources]# ./flock -f -w Flock.c
Lock acquired
```

che ci mostra come i due tipi di blocco siano assolutamente indipendenti; per questo motivo occorre sempre tenere presente quale, fra le due semantiche disponibili, stanno usando i programmi con cui si interagisce, dato che i blocchi applicati con l'altra non avrebbero nessun effetto.

Abbiamo visto come l'interfaccia POSIX per il *file locking* sia molto più potente e flessibile di quella di BSD, questo comporta anche una maggiore complessità per via delle varie opzioni da passare a `fcntl`. Per questo motivo è disponibile anche una interfaccia semplificata che utilizza la funzione `lockf`,⁷ il cui prototipo è:

<pre>#include <unistd.h> int lockf(int fd, int cmd, off_t len)</pre>		Applica, controlla o rimuove un <i>file lock</i> .
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:		
EAGAIN	il file è bloccato, e si sono richiesti F_TLOCK o F_TEST (in alcuni casi può dare anche EACCESS.	
EBADF	fd non è un file descriptor aperto o si sono richiesti F_LOCK o F_TLOCK ma il file non è scrivibile.	
EINVAL	si è usato un valore non valido per cmd.	
ed inoltre EDEADLK e ENOLCK con lo stesso significato che hanno con <code>fcntl</code> .		

La funzione opera sul file indicato dal file descriptor `fd`, che deve essere aperto in scrittura, perché utilizza soltanto *lock* esclusivi. La sezione di file bloccata viene controllata dal valore di `len`, che indica la lunghezza della stessa, usando come riferimento la posizione corrente sul file. La sezione effettiva varia a secondo del segno, secondo lo schema illustrato in fig. 10.6, se si specifica un valore nullo il file viene bloccato a partire dalla posizione corrente fino alla sua fine presente o futura (nello schema corrisponderebbe ad un valore infinito positivo).

Figura 10.6: Schema della sezione di file bloccata con `lockf`.

Il comportamento della funzione viene controllato dal valore dell'argomento `cmd`, che specifica quale azione eseguire, i soli valori consentiti sono i seguenti:

F_LOCK Richiede un *lock* esclusivo sul file, e blocca il processo chiamante se, anche parzialmente, la sezione indicata si sovrappone ad una che è già stata bloccata da un altro processo; in caso di sovrapposizione con un altro blocco già ottenuto le sezioni vengono unite.

⁷la funzione è ripresa da System V e per poterla utilizzare è richiesta che siano definite le opportune macro, una fra `_BSD_SOURCE` o `_SVID_SOURCE`, oppure `_XOPEN_SOURCE` ad un valore di almeno 500, oppure `_XOPEN_SOURCE` e `_XOPEN_SOURCE_EXTENDED`.

F_TLOCK	Richiede un <i>exclusive lock</i> , in maniera identica a F_LOCK, ma in caso di indisponibilità non blocca il processo restituendo un errore di EAGAIN.
F_ULOCK	Rilascia il blocco sulla sezione indicata, questo può anche causare la suddivisione di una sezione bloccata in precedenza nelle due parti eccedenti nel caso si sia indicato un intervallo più limitato.
F_TEST	Controlla la presenza di un blocco sulla sezione di file indicata, <code>lockf</code> ritorna 0 se la sezione è libera o bloccata dal processo stesso, o -1 se è bloccata da un altro processo, nel qual caso <code>errno</code> assume il valore EAGAIN (ma su alcuni sistemi può essere restituito anche EACCESS).

La funzione è semplicemente una diversa interfaccia al *file locking* POSIX ed è realizzata utilizzando `fcntl`; pertanto la semantica delle operazioni è la stessa di quest'ultima e quindi la funzione presenta lo stesso comportamento riguardo gli effetti della chiusura dei file, ed il comportamento sui file duplicati e nel passaggio attraverso `fork` ed `exec`. Per questo stesso motivo la funzione non è equivalente a `flock` e può essere usata senza interferenze insieme a quest'ultima.

10.1.4 Gli *open file descriptor locks*

Come illustrato in dettaglio nella precedente sez. 10.1.3, la chiusura di un file su cui sono presenti dei *file lock* comporta l'immediato rilascio degli stessi, anche se questi sono stati acquisiti da un processo diverso.

da finire.

10.1.5 Il *mandatory locking*

Il *mandatory locking* è una opzione introdotta inizialmente in SVr4, per introdurre un *file locking* che, come dice il nome, fosse effettivo indipendentemente dai controlli eseguiti da un processo. Con il *mandatory locking* infatti è possibile far eseguire il blocco del file direttamente al sistema, così che, anche qualora non si predisponessero le opportune verifiche nei processi, questo verrebbe comunque rispettato.

Per poter utilizzare il *mandatory locking* è stato introdotto un utilizzo particolare del bit *sgid* dei permessi dei file. Se si ricorda quanto esposto in sez. 4.4.2), esso viene di norma utilizzato per cambiare il *GID* effettivo con cui viene eseguito un programma, ed è pertanto sempre associato alla presenza del permesso di esecuzione per il gruppo. Impostando questo bit su un file senza permesso di esecuzione in un sistema che supporta il *mandatory locking*, fa sì che quest'ultimo venga attivato per il file in questione. In questo modo una combinazione dei permessi originariamente non contemplata, in quanto senza significato, diventa l'indicazione della presenza o meno del *mandatory locking*.⁸

L'uso del *mandatory locking* presenta vari aspetti delicati, dato che neanche l'amministratore può passare sopra ad un *file lock*; pertanto un processo che blocchi un file cruciale può renderlo completamente inaccessibile, rendendo completamente inutilizzabile il sistema⁹ inoltre con il *mandatory locking* si può bloccare completamente un server NFS richiedendo

⁸un lettore attento potrebbe ricordare quanto detto in sez. 4.4.3 e cioè che il bit *sgid* viene cancellato (come misura di sicurezza) quando si scrive su un file, questo non vale quando esso viene utilizzato per attivare il *mandatory locking*.

⁹il problema si potrebbe risolvere rimuovendo il bit *sgid*, ma non è detto che sia così facile fare questa operazione con un sistema bloccato.

una lettura su un file su cui è attivo un blocco. Per questo motivo l'abilitazione del *mandatory locking* è di norma disabilitata, e deve essere attivata filesystem per filesystem in fase di montaggio, specificando l'apposita opzione di `mount` riportata in sez. 4.1.4, o con l'opzione `-o mand` per il comando omonimo.

Si tenga presente inoltre che il *mandatory locking* funziona solo sull'interfaccia POSIX di `fcntl`. Questo ha due conseguenze: che non si ha nessun effetto sui *file lock* richiesti con l'interfaccia di `flock`, e che la granularità del blocco è quella del singolo byte, come per `fcntl`.

La sintassi di acquisizione dei blocchi è esattamente la stessa vista in precedenza per `fcntl` e `lockf`, la differenza è che in caso di *mandatory lock* attivato non è più necessario controllare la disponibilità di accesso al file, ma si potranno usare direttamente le ordinarie funzioni di lettura e scrittura e sarà compito del kernel gestire direttamente il *file locking*.

Questo significa che in caso di *read lock* la lettura dal file potrà avvenire normalmente con `read`, mentre una *write* si bloccherà fino al rilascio del blocco, a meno di non aver aperto il file con `O_NONBLOCK`, nel qual caso essa ritornerà immediatamente con un errore di `EAGAIN`.

Se invece si è acquisito un *write lock* tutti i tentativi di leggere o scrivere sulla regione del file bloccata fermeranno il processo fino al rilascio del blocco, a meno che il file non sia stato aperto con `O_NONBLOCK`, nel qual caso di nuovo si otterrà un ritorno immediato con l'errore di `EAGAIN`.

Infine occorre ricordare che le funzioni di lettura e scrittura non sono le sole ad operare sui contenuti di un file, e che sia `creat` che `open` (quando chiamata con `O_TRUNC`) effettuano dei cambiamenti, così come `truncate`, riducendone le dimensioni (a zero nei primi due casi, a quanto specificato nel secondo). Queste operazioni sono assimilate a degli accessi in scrittura e pertanto non potranno essere eseguite (fallendo con un errore di `EAGAIN`) su un file su cui sia presente un qualunque blocco (le prime due sempre, la terza solo nel caso che la riduzione delle dimensioni del file vada a sovrapporsi ad una regione bloccata).

L'ultimo aspetto della interazione del *mandatory locking* con le funzioni di accesso ai file è quello relativo ai file mappati in memoria (vedi sez. 10.4.1); anche in tal caso infatti, quando si esegue la mappatura con l'opzione `MAP_SHARED`, si ha un accesso al contenuto del file. Lo standard SVID prevede che sia impossibile eseguire il *memory mapping* di un file su cui sono presenti dei blocchi¹⁰ in Linux è stata però fatta la scelta implementativa¹¹ di seguire questo comportamento soltanto quando si chiama `mmap` con l'opzione `MAP_SHARED` (nel qual caso la funzione fallisce con il solito `EAGAIN`) che comporta la possibilità di modificare il file.

Si tenga conto infine che su Linux l'implementazione corrente del *mandatory locking* è difettosa e soffre di una *race condition*, per cui una scrittura con `write` che si sovrapponga alla richiesta di un *read lock* può modificare i dati anche dopo che questo è stato ottenuto, ed una lettura con `read` può restituire dati scritti dopo l'ottenimento di un *write lock*. Lo stesso tipo di problema si può presentare anche con l'uso di file mappati in memoria; pertanto allo stato attuale delle cose è sconsigliabile fare affidamento sul *mandatory locking*.

10.2 L'I/O multiplexing

Uno dei problemi che si presentano quando si deve operare contemporaneamente su molti file usando le funzioni illustrate in sez. 5.1 e sez. 5.3 è che si può essere bloccati nelle operazioni su un file mentre un altro potrebbe essere disponibile. L'I/O *multiplexing* nasce risposta a

¹⁰alcuni sistemi, come HP-UX, sono ancora più restrittivi e lo impediscono anche in caso di *advisory locking*, anche se questo comportamento non ha molto senso, dato che comunque qualunque accesso diretto al file è consentito.

¹¹per i dettagli si possono leggere le note relative all'implementazione, mantenute insieme ai sorgenti del kernel nel file `Documentation/mandatory.txt`.

questo problema. In questa sezione forniremo una introduzione a questa problematica ed analizzeremo le varie funzioni usate per implementare questa modalità di I/O.

10.2.1 La problematica dell'I/O *multiplexing*

Abbiamo visto in sez. 7.3.1, affrontando la suddivisione fra *fast* e *slow system call*, che in certi casi le funzioni di I/O eseguite su un file descriptor possono bloccarsi indefinitamente. Questo non avviene mai per i file normali, per i quali le funzioni di lettura e scrittura ritornano sempre subito, ma può avvenire per alcuni file di dispositivo, come ad esempio una seriale o un terminale, o con l'uso di file descriptor collegati a meccanismi di intercomunicazione come le *pipe* (vedi sez. 11.1) ed i socket (vedi sez. 14.1.1). In casi come questi ad esempio una operazione di lettura potrebbe bloccarsi se non ci sono dati disponibili sul descrittore su cui la si sta effettuando.

Questo comportamento è alla radice di una delle problematiche più comuni che ci si trova ad affrontare nella gestione delle operazioni di I/O: la necessità di operare su più file descriptor eseguendo funzioni che possono bloccarsi indefinitamente senza che sia possibile prevedere quando questo può avvenire. Un caso classico è quello di un server di rete (tratteremo la problematica in dettaglio nella seconda parte della guida) in attesa di dati in ingresso provenienti da vari client.

In un caso di questo tipo, se si andasse ad operare sui vari file descriptor aperti uno dopo l'altro, potrebbe accadere di restare bloccati nell'eseguire una lettura su uno di quelli che non è “pronto”, quando ce ne potrebbe essere un altro con dati disponibili. Questo comporta nel migliore dei casi una operazione ritardata inutilmente nell'attesa del completamento di quella bloccata, mentre nel peggiore dei casi, quando la conclusione dell'operazione bloccata dipende da quanto si otterrebbe dal file descriptor “disponibile”, si potrebbe addirittura arrivare ad un *deadlock*.

Abbiamo già accennato in sez. 5.1.2 che è possibile prevenire questo tipo di comportamento delle funzioni di I/O aprendo un file in *modalità non-bloccante*, attraverso l'uso del flag `O_NONBLOCK` nella chiamata di `open`. In questo caso le funzioni di lettura o scrittura eseguite sul file che si sarebbero bloccate ritornano immediatamente, restituendo l'errore `EAGAIN`. L'utilizzo di questa modalità di I/O permette di risolvere il problema controllando a turno i vari file descriptor, in un ciclo in cui si ripete l'accesso fintanto che esso non viene garantito. Ovviamente questa tecnica, detta *polling*, è estremamente inefficiente: si tiene costantemente impiegata la CPU solo per eseguire in continuazione delle *system call* che nella gran parte dei casi falliranno.

È appunto per superare questo problema è stato introdotto il concetto di *I/O multiplexing*, una nuova modalità per la gestione dell'I/O che consente di tenere sotto controllo più file descriptor in contemporanea, permettendo di bloccare un processo quando le operazioni di lettura o scrittura non sono immediatamente effettuabili, e di riprenderne l'esecuzione una volta che almeno una di quelle che erano state richieste diventi possibile, in modo da poterla eseguire con la sicurezza di non restare bloccati.

Dato che, come abbiamo già accennato, per i normali file su disco non si ha mai un accesso bloccante, l'uso più comune delle funzioni che esamineremo nei prossimi paragrafi è per i server di rete, in cui esse vengono utilizzate per tenere sotto controllo dei socket; pertanto ritorneremo su di esse con ulteriori dettagli e qualche esempio di utilizzo concreto in sez. 15.6.

10.2.2 Le funzioni `select` e `pselect`

Il primo kernel unix-like ad introdurre una interfaccia per l'*I/O multiplexing* è stato BSD, con la funzione `select` che è apparsa in BSD4.2 ed è stata standardizzata in BSD4.4, in seguito è stata portata su tutti i sistemi che supportano i socket, compreso le varianti di System V ed inserita in POSIX.1-2001; il suo prototipo è:¹²

```
#include <sys/select.h>
int select(int ndfs, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout)
           Attende che uno fra i file descriptor degli insiemi specificati diventi attivo.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF si è specificato un file descriptor non valido (chiuso o con errori) in uno degli insiemi.
EINTR la funzione è stata interrotta da un segnale.
EINVAL si è specificato per `ndfs` un valore negativo o un valore non valido per `timeout`.
ed inoltre **ENOMEM** nel suo significato generico.

La funzione mette il processo in stato di *sleep* (vedi tab. 3.8) fintanto che almeno uno dei file descriptor degli insiemi specificati (`readfds`, `writefds` e `exceptfds`), non diventa attivo, per un tempo massimo specificato da `timeout`.

Per specificare quali file descriptor si intende selezionare la funzione usa un particolare oggetto, il *file descriptor set*, identificato dal tipo `fd_set`, che serve ad identificare un insieme di file descriptor, in maniera analoga a come un *signal set* (vedi sez. 7.4.2) identifica un insieme di segnali. Per la manipolazione di questi *file descriptor set* si possono usare delle opportune macro di preprocessore:

```
#include <sys/select.h>
void FD_ZERO(fd_set *set)
                                     Inizializza l'insieme (vuoto).

void FD_SET(int fd, fd_set *set)
                                     Inserisce il file descriptor fd nell'insieme.

void FD_CLR(int fd, fd_set *set)
                                     Rimuove il file descriptor fd dall'insieme.

int FD_ISSET(int fd, fd_set *set)
                                     Controlla se il file descriptor fd è nell'insieme.
```

In genere un *file descriptor set* può contenere fino ad un massimo di `FD_SETSIZE` file descriptor. Questo valore in origine corrispondeva al limite per il numero massimo di file aperti (ad esempio in Linux, fino alla serie 2.0.x, c'era un limite di 256 file per processo), ma da quando, nelle versioni più recenti del kernel, questo limite è stato rimosso, esso indica le dimensioni massime dei numeri usati nei *file descriptor set*, ed il suo valore, secondo lo standard POSIX 1003.1-2001, è definito in `sys/select.h`, ed è pari a 1024.

Si tenga presente che i *file descriptor set* devono sempre essere inizializzati con `FD_ZERO`; passare a `select` un valore non inizializzato può dar luogo a comportamenti non prevedibili. Allo stesso modo usare `FD_SET` o `FD_CLR` con un file descriptor il cui valore eccede `FD_SETSIZE` può dare luogo ad un comportamento indefinito.

La funzione richiede di specificare tre insiemi distinti di file descriptor; il primo, `readfds`, verrà osservato per rilevare la disponibilità di effettuare una lettura,¹³ il secondo, `writefds`,

¹²l'header `sys/select.h` è stato introdotto con POSIX.1-2001, è ed presente con la *glibc* a partire dalla versione 2.0, in precedenza, con le *libc4* e *libc5*, occorreva includere `sys/time.h`, `sys/types.h` e `unistd.h`.

¹³per essere precisi la funzione ritornerà in tutti i casi in cui la successiva esecuzione di `read` risulti non bloccante, quindi anche in caso di *end-of-file*.

per verificare la possibilità di effettuare una scrittura ed il terzo, `exceptfds`, per verificare l'esistenza di eccezioni come i dati urgenti su un socket, (vedi sez. 18.1.4).

Dato che in genere non si tengono mai sotto controllo fino a `FD_SETSIZE` file contemporaneamente, la funzione richiede di specificare qual è il valore più alto fra i file descriptor indicati nei tre insiemi precedenti. Questo viene fatto per efficienza, per evitare di passare e far controllare al kernel una quantità di memoria superiore a quella necessaria. Questo limite viene indicato tramite l'argomento `ndfs`, che deve corrispondere al valore massimo aumentato di uno. Si ricordi infatti che i file descriptor sono numerati progressivamente a partire da zero, ed il valore indica il numero più alto fra quelli da tenere sotto controllo, dimenticarsi di aumentare di uno il valore di `ndfs` è un errore comune.

Infine l'argomento `timeout`, espresso con il puntatore ad una struttura di tipo `timeval` (vedi fig. 4.15) specifica un tempo massimo di attesa prima che la funzione ritorni; se impostato a `NULL` la funzione attende indefinitamente. Si può specificare anche un tempo nullo (cioè una struttura `timeval` con i campi impostati a zero), qualora si voglia semplicemente controllare lo stato corrente dei file descriptor, e così può essere utilizzata eseguire il *polling* su un gruppo di file descriptor. Usare questo argomento con tutti i *file descriptor set* vuoti è un modo portabile, disponibile anche su sistemi in cui non sono disponibili le funzioni avanzate di sez. 7.5.2, per tenere un processo in stato di *sleep* con precisioni inferiori al secondo.

In caso di successo la funzione restituisce il numero di file descriptor pronti, seguendo il comportamento previsto dallo standard POSIX.1-2001,¹⁴ e ciascun insieme viene sovrascritto per indicare quali sono i file descriptor pronti per le operazioni ad esso relative, in modo da poterli controllare con `FD_ISSET`. Se invece scade il tempo indicato da `timeout` viene restituito un valore nullo e i *file descriptor set* non vengono modificati. In caso di errore la funzione restituisce `-1`, i valori dei tre insiemi e di `timeout` sono indefiniti e non si può fare nessun affidamento sul loro contenuto; nelle versioni più recenti della funzione invece i *file descriptor set* non vengono modificati anche in caso di errore.

Si tenga presente infine che su Linux, in caso di programmazione *multi-thread* se un file descriptor viene chiuso in un altro *thread* rispetto a quello in cui si sta usando `select`, questa non subisce nessun effetto. In altre varianti di sistemi unix-like invece `select` ritorna indicando che il file descriptor è pronto, con conseguente possibile errore nel caso lo si usi senza che sia stato riaperto. Lo standard non prevede niente al riguardo e non si deve dare per assunto nessuno dei due comportamenti se si vogliono scrivere programmi portabili.

Una volta ritornata la funzione, si potrà controllare quali sono i file descriptor pronti, ed operare su di essi. Si tenga presente però che `select` fornisce solo di un suggerimento, esistono infatti condizioni in cui `select` può riportare in maniera spuria che un file descriptor è pronto, ma l'esecuzione di una operazione di I/O si bloccherebbe: ad esempio con Linux questo avviene quando su un socket arrivano dei dati che poi vengono scartati perché corrotti (ma sono possibili pure altri casi); in tal caso pur risultando il relativo file descriptor pronto in lettura una successiva esecuzione di una `read` si bloccherebbe. Per questo motivo quando si usa l'I/O *multiplexing* è sempre raccomandato l'uso delle funzioni di lettura e scrittura in modalità non bloccante.

Su Linux quando la *system call* `select` viene interrotta da un segnale modifica il valore nella struttura puntata da `timeout`, impostandolo al tempo restante. In tal caso infatti si ha un errore di `EINTR` ed occorre rilanciare la funzione per proseguire l'attesa, ed in questo modo non è necessario ricalcolare tutte le volte il tempo rimanente. Questo può causare problemi di portabilità sia quando si usa codice scritto su Linux che legge questo valore, sia quando si usano programmi scritti per altri sistemi che non dispongono di questa caratteristica e

¹⁴si tenga però presente che esistono alcune versioni di Unix che non si comportano in questo modo, restituendo un valore positivo generico.

ricalcolano `timeout` tutte le volte. In genere questa caratteristica è disponibile nei sistemi che derivano da System V e non è disponibile per quelli che derivano da BSD; lo standard POSIX.1-2001 non permette questo comportamento e per questo motivo la *glibc* nasconde il comportamento passando alla *system call* una copia dell'argomento `timeout`.

Uno dei problemi che si presentano con l'uso di `select` è che il suo comportamento dipende dal valore del file descriptor che si vuole tenere sotto controllo. Infatti il kernel riceve con `ndfs` un limite massimo per tale valore, e per capire quali sono i file descriptor da tenere sotto controllo dovrà effettuare una scansione su tutto l'intervallo, che può anche essere molto ampio anche se i file descriptor sono solo poche unità; tutto ciò ha ovviamente delle conseguenze ampiamente negative per le prestazioni.

Inoltre c'è anche il problema che il numero massimo dei file che si possono tenere sotto controllo, la funzione è nata quando il kernel consentiva un numero massimo di 1024 file descriptor per processo, adesso che il numero può essere arbitrario si viene a creare una dipendenza del tutto artificiale dalle dimensioni della struttura `fd_set`, che può necessitare di essere estesa, con ulteriori perdite di prestazioni.

Lo standard POSIX è rimasto a lungo senza primitive per l'*I/O multiplexing*, introdotto solo con le ultime revisioni dello standard (POSIX 1003.1g-2000 e POSIX 1003.1-2001). La scelta è stata quella di seguire l'interfaccia creata da BSD, ma prevede che tutte le funzioni ad esso relative vengano dichiarate nell'header `sys/select.h`, che sostituisce i precedenti, ed inoltre aggiunge a `select` una nuova funzione `pselect`,¹⁵ il cui prototipo è:

```
#include <sys/select.h>
int pselect(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timespec *timeout, sigset_t *sigmask)
    Attende che uno dei file descriptor degli insiemi specificati diventi attivo.
```

La funzione ritorna il numero (anche nullo) di file descriptor che sono attivi in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` si è specificato un file descriptor sbagliato in uno degli insiemi.

`EINTR` la funzione è stata interrotta da un segnale.

`EINVAL` si è specificato per `ndfs` un valore negativo o un valore non valido per `timeout`. ed inoltre `ENOMEM` nel suo significato generico.

La funzione è sostanzialmente identica a `select`, solo che usa una struttura `timespec` (vedi fig. 4.16) per indicare con maggiore precisione il `timeout` e non ne aggiorna il valore in caso di interruzione. In realtà anche in questo caso la *system call* di Linux aggiorna il valore al tempo rimanente, ma la funzione fornita dalla *glibc* modifica questo comportamento passando alla *system call* una variabile locale, in modo da mantenere l'aderenza allo standard POSIX che richiede che il valore di `timeout` non sia modificato.

Rispetto a `select` la nuova funzione prende un argomento aggiuntivo `sigmask`, un puntatore ad una maschera di segnali (si veda sez. 7.4.4). Nell'esecuzione la maschera dei segnali corrente viene sostituita da quella così indicata immediatamente prima di eseguire l'attesa, e viene poi ripristinata al ritorno della funzione. L'uso di `sigmask` è stato introdotto allo scopo di prevenire possibili *race condition* quando oltre alla presenza di dati sui file descriptor come nella `select` ordinaria, ci si deve porre in attesa anche dell'arrivo di un segnale.

Come abbiamo visto in sez. 7.4.1 la tecnica classica per rilevare l'arrivo di un segnale è quella di utilizzare il gestore per impostare una variabile globale e controllare questa nel corpo principale del programma; abbiamo visto in quell'occasione come questo lasci spazio a

¹⁵il supporto per lo standard POSIX 1003.1-2001, ed l'header `sys/select.h`, compaiono in Linux a partire dalla *glibc* 2.1. Le *libc4* e *libc5* non contengono questo header, la *glibc* 2.0 contiene una definizione sbagliata di `psignal`, senza l'argomento `sigmask`, la definizione corretta è presente dalle *glibc* 2.1-2.2.1 se si è definito `_GNU_SOURCE` e nelle *glibc* 2.2.2-2.2.4 se si è definito `_XOPEN_SOURCE` con valore maggiore di 600.

possibili *race condition*, per cui diventa essenziale utilizzare `sigprocmask` per disabilitare la ricezione del segnale prima di eseguire il controllo e riabilitarlo dopo l'esecuzione delle relative operazioni, onde evitare l'arrivo di un segnale immediatamente dopo il controllo, che andrebbe perso.

Nel nostro caso il problema si pone quando, oltre al segnale, si devono tenere sotto controllo anche dei file descriptor con `select`, in questo caso si può fare conto sul fatto che all'arrivo di un segnale essa verrebbe interrotta e si potrebbero eseguire di conseguenza le operazioni relative al segnale e alla gestione dati con un ciclo del tipo:

```
while (1) {
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    if (receive_signal != 0) handle_signal();
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    n = select(nfd, rset, wset, eset, NULL);
    if (n < 0) {
        if (errno == EINTR) {
            continue;
        }
    } else handle_filedata();
}
```

qui però emerge una *race condition*, perché se il segnale arriva prima della chiamata a `select`, questa non verrà interrotta, e la ricezione del segnale non sarà rilevata.

Per questo è stata introdotta `pselect` che attraverso l'argomento `sigmask` permette di riabilitare la ricezione il segnale contestualmente all'esecuzione della funzione,¹⁶ ribloccandolo non appena essa ritorna, così che il precedente codice potrebbe essere riscritto nel seguente modo:

```
while (1) {
    sigprocmask(SIG_BLOCK, &newmask, &oldmask);
    if (receive_signal != 0) handle_signal();
    n = pselect(nfd, rset, wset, eset, NULL, &oldmask);
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    if (n < 0) {
        if (errno == EINTR) {
            continue;
        }
    } else {
        handle_filedata();
    }
}
```

in questo caso utilizzando `oldmask` durante l'esecuzione di `pselect` la ricezione del segnale sarà abilitata, ed in caso di interruzione si potranno eseguire le relative operazioni.

¹⁶in Linux però, fino al kernel 2.6.16, non era presente la relativa *system call*, e la funzione era implementata nella *glibc* attraverso `select` (vedi `man select_tut`) per cui la possibilità di *race condition* permaneva; in tale situazione si può ricorrere ad una soluzione alternativa, chiamata *self-pipe trick*, che consiste nell'aprire una *pipe* (vedi sez. 11.1.1) ed usare `select` sul capo in lettura della stessa; si può indicare l'arrivo di un segnale scrivendo sul capo in scrittura all'interno del gestore dello stesso; in questo modo anche se il segnale va perso prima della chiamata di `select` questa lo riconoscerà comunque dalla presenza di dati sulla *pipe*.

10.2.3 Le funzioni poll e ppoll

Nello sviluppo di System V, invece di utilizzare l'interfaccia di **select**, che è una estensione tipica di BSD, è stata introdotta una interfaccia completamente diversa, basata sulla funzione di sistema **poll**,¹⁷ il cui prototipo è:

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, nfds_t nfd, int timeout)
    Attende un cambiamento di stato su un insieme di file descriptor.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF si è specificato un file descriptor sbagliato in uno degli insiemi.
 EINTR la funzione è stata interrotta da un segnale.
 EINVAL il valore di `nfd` eccede il limite `RLIMIT_NOFILE`.
 ed inoltre EFAULT e ENOMEM nel loro significato generico.

La funzione permette di tenere sotto controllo contemporaneamente `nfd` file descriptor, specificati attraverso il puntatore `ufds` ad un vettore di strutture `pollfd`. Come con **select** si può interrompere l'attesa dopo un certo tempo, questo deve essere specificato con l'argomento `timeout` in numero di millisecondi: un valore negativo indica un'attesa indefinita, mentre un valore nullo comporta il ritorno immediato, e può essere utilizzato per impiegare **poll** in modalità *non-bloccante*.

```
struct pollfd {
    int fd;           /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

Figura 10.7: La struttura `pollfd`, utilizzata per specificare le modalità di controllo di un file descriptor alla funzione **poll**.

Per ciascun file da controllare deve essere inizializzata una struttura `pollfd` nel vettore indicato dall'argomento `ufds`. La struttura, la cui definizione è riportata in fig. 10.7, prevede tre campi: in `fd` deve essere indicato il numero del file descriptor da controllare, in `events` deve essere specificata una maschera binaria di flag che indichino il tipo di evento che si vuole controllare, mentre in `revents` il kernel restituirà il relativo risultato.

Usando un valore negativo per `fd` la corrispondente struttura sarà ignorata da **poll** ed il campo `revents` verrà azzerato, questo consente di eliminare temporaneamente un file descriptor dalla lista senza dover modificare il vettore `ufds`. Dato che i dati in ingresso sono del tutto indipendenti da quelli in uscita (che vengono restituiti in `revents`) non è necessario reinizializzare tutte le volte il valore delle strutture `pollfd` a meno di non voler cambiare qualche condizione.

Le costanti che definiscono i valori relativi ai bit usati nelle maschere binarie dei campi `events` e `revents` sono riportate in tab. 10.4, insieme al loro significato. Le si sono suddivise in tre gruppi principali, nel primo gruppo si sono indicati i bit utilizzati per controllare l'attività

¹⁷la funzione è prevista dallo standard XPG4, ed è stata introdotta in Linux come *system call* a partire dal kernel 2.1.23 ed inserita nelle *libc* 5.4.28, originariamente l'argomento `nfd` era di tipo `unsigned int`, la funzione è stata inserita nello standard POSIX.1-2001 in cui è stato introdotto il tipo nativo `nfds_t`.

in ingresso, nel secondo quelli per l'attività in uscita, infine il terzo gruppo contiene dei valori che vengono utilizzati solo nel campo `revents` per notificare delle condizioni di errore.

Flag	Significato
POLLIN	È possibile la lettura.
POLLRDNORM	Sono disponibili in lettura dati normali.
POLLRDBAND	Sono disponibili in lettura dati prioritari.
POLLPRI	È possibile la lettura di dati urgenti.
POLLOUT	È possibile la scrittura immediata.
POLLWRNORM	È possibile la scrittura di dati normali.
POLLWRBAND	È possibile la scrittura di dati prioritari.
POLLERR	C'è una condizione di errore.
POLLHUP	Si è verificato un hung-up.
POLLRDHUP	Si è avuta una <i>half-close</i> su un socket. ¹⁸
POLLNVAL	Il file descriptor non è aperto.
POLLMSG	Definito per compatibilità con SysV.

Tabella 10.4: Costanti per l'identificazione dei vari bit dei campi `events` e `revents` di `pollfd`.

Il valore `POLLMSG` non viene utilizzato ed è definito solo per compatibilità con l'implementazione di System V che usa i cosiddetti “*stream*”. Si tratta di una interfaccia specifica di SysV non presente in Linux, che non ha nulla a che fare con gli *stream* delle librerie standard del C visti in sez. 5.3.1. Da essa derivano i nomi di alcune costanti poiché per quegli *stream* sono definite tre classi di dati: *normali*, *prioritari* ed *urgenti*. In Linux la distinzione ha senso solo per i dati urgenti dei socket (vedi sez. 18.1.4), ma su questo e su come `poll` reagisce alle varie condizioni dei socket torneremo in sez. 15.6.5, dove vedremo anche un esempio del suo utilizzo.

Le costanti relative ai diversi tipi di dati normali e prioritari che fanno riferimento alle implementazioni in stile System V sono `POLLRDNORM`, `POLLWRNORM`, `POLLRDBAND` e `POLLWRBAND`. Le prime due sono equivalenti rispettivamente a `POLLIN` e `POLLOUT`, `POLLRDBAND` non viene praticamente mai usata su Linux mentre `POLLWRBAND` ha senso solo sui socket. In ogni caso queste costanti sono utilizzabili soltanto qualora si sia definita la macro `_XOPEN_SOURCE`.

In caso di successo `poll` ritorna restituendo il numero di file (un valore positivo) per i quali si è verificata una delle condizioni di attesa richieste o per i quali si è verificato un errore, avvalorando i relativi bit di `revents`. In caso di errori sui file vengono utilizzati i valori della terza sezione di tab. 10.4 che hanno significato solo per `revents` (se specificati in `events` vengono ignorati). Un valore di ritorno nullo indica che si è raggiunto il timeout, mentre un valore negativo indica un errore nella chiamata, il cui codice viene riportato al solito tramite `errno`.

L'uso di `poll` consente di superare alcuni dei problemi illustrati in precedenza per `select`; anzitutto, dato che in questo caso si usa un vettore di strutture `pollfd` di dimensione arbitraria, non esiste il limite introdotto dalle dimensioni massime di un *file descriptor set* e la dimensione dei dati passati al kernel dipende solo dal numero dei file descriptor che si vogliono controllare, non dal loro valore. Infatti, anche se usando dei bit un *file descriptor set* può essere più efficiente di un vettore di strutture `pollfd`, qualora si debba osservare un solo file descriptor con un valore molto alto ci si troverà ad utilizzare inutilmente un maggiore quantitativo di memoria.

Inoltre con `select` lo stesso *file descriptor set* è usato sia in ingresso che in uscita, e questo significa che tutte le volte che si vuole ripetere l'operazione occorre reinizializzarlo da

¹⁸Si tratta di una estensione specifica di Linux, disponibile a partire dal kernel 2.6.17 definendo la macro `_GNU_SOURCE`, che consente di riconoscere la chiusura in scrittura dell'altro capo di un socket, situazione che si viene chiamata appunto *half-close* (mezza chiusura) su cui torneremo con maggiori dettagli in sez. 15.6.3.

capo. Questa operazione, che può essere molto onerosa se i file descriptor da tenere sotto osservazione sono molti, non è invece necessaria con `poll`.

Abbiamo visto in sez. 10.2.2 come lo standard POSIX preveda una variante di `select` che consente di gestire correttamente la ricezione dei segnali nell'attesa su un file descriptor. Con l'introduzione di una implementazione reale di `pselect` nel kernel 2.6.16, è stata aggiunta anche una analoga funzione che svolga lo stesso ruolo per `poll`.

In questo caso si tratta di una estensione che è specifica di Linux e non è prevista da nessuno standard; essa può essere utilizzata esclusivamente se si definisce la macro `_GNU_SOURCE` ed ovviamente non deve essere usata se si ha a cuore la portabilità. La funzione è `ppoll`, ed il suo prototipo è:

```
#include <sys/poll.h>
int ppoll(struct pollfd *fds, nfds_t nfds, const struct timespec *timeout,
          const sigset_t *sigmask)
    Attende un cambiamento di stato su un insieme di file descriptor.
```

La funzione ritorna il numero di file descriptor con attività in caso di successo, 0 se c'è stato un timeout e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` si è specificato un file descriptor sbagliato in uno degli insiemi.
`EINTR` la funzione è stata interrotta da un segnale.
`EINVAL` il valore di `nfds` eccede il limite `RLIMIT_NOFILE`.
ed inoltre `EFAULT` e `ENOMEM` nel loro significato generico.

La funzione ha lo stesso comportamento di `poll`, solo che si può specificare, con l'argomento `sigmask`, il puntatore ad una maschera di segnali; questa sarà la maschera utilizzata per tutto il tempo che la funzione resterà in attesa, all'uscita viene ripristinata la maschera originale. L'uso di questa funzione è cioè equivalente, come illustrato nella pagina di manuale, all'esecuzione atomica del seguente codice:

```
sigset_t origmask;
sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = poll(&fds, nfds, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);
```

Eccetto per `timeout`, che come per `pselect` deve essere un puntatore ad una struttura `timespec`, gli altri argomenti comuni con `poll` hanno lo stesso significato, e la funzione restituisce gli stessi risultati illustrati in precedenza. Come nel caso di `pselect` la *system call* che implementa `ppoll` restituisce, se la funzione viene interrotta da un segnale, il tempo mancante in `timeout`, e come per `pselect` la funzione di libreria fornita dalla *glibc* maschera questo comportamento non modificando mai il valore di `timeout` anche se in questo caso non esiste nessuno standard che richieda questo comportamento.

Infine anche per `poll` e `ppoll` valgono le considerazioni relative alla possibilità di avere delle notificazioni spurie della disponibilità di accesso ai file descriptor illustrate per `select` in sez. 10.2.2, che non staremo a ripetere qui.

10.2.4 L'interfaccia di *epoll*

Nonostante `poll` presenti alcuni vantaggi rispetto a `select`, anche questa funzione non è molto efficiente quando deve essere utilizzata con un gran numero di file descriptor,¹⁹ in particolare nel caso in cui solo pochi di questi diventano attivi. Il problema in questo caso è che il tempo

¹⁹in casi del genere `select` viene scartata a priori, perché può avvenire che il numero di file descriptor ecceda le dimensioni massime di un *file descriptor set*.

impiegato da `poll` a trasferire i dati da e verso il kernel è proporzionale al numero di file descriptor osservati, non a quelli che presentano attività.

Quando ci sono decine di migliaia di file descriptor osservati e migliaia di eventi al secondo (il caso classico è quello di un server web di un sito con molti accessi) l'uso di `poll` comporta la necessità di trasferire avanti ed indietro da *user space* a *kernel space* una lunga lista di strutture `pollfd` migliaia di volte al secondo. A questo poi si aggiunge il fatto che la maggior parte del tempo di esecuzione sarà impegnato ad eseguire una scansione su tutti i file descriptor tenuti sotto controllo per determinare quali di essi (in genere una piccola percentuale) sono diventati attivi. In una situazione come questa l'uso delle funzioni classiche dell'interfaccia dell'I/O *multiplexing* viene a costituire un collo di bottiglia che degrada irrimediabilmente le prestazioni.

Per risolvere questo tipo di situazioni sono state ideate delle interfacce specialistiche (come `/dev/poll` in Solaris, o `kqueue` in BSD) il cui scopo fondamentale è quello di restituire solamente le informazioni relative ai file descriptor osservati che presentano una attività, evitando così le problematiche appena illustrate. In genere queste prevedono che si registrino una sola volta i file descriptor da tenere sotto osservazione, e forniscono un meccanismo che notifica quali di questi presentano attività.

Le modalità con cui avviene la notifica sono due, la prima è quella classica (quella usata da `poll` e `select`) che viene chiamata *level triggered*.²⁰ In questa modalità vengono notificati i file descriptor che sono *pronti* per l'operazione richiesta, e questo avviene indipendentemente dalle operazioni che possono essere state fatte su di essi a partire dalla precedente notifica. Per chiarire meglio il concetto ricorriamo ad un esempio: se su un file descriptor sono diventati disponibili in lettura 2000 byte ma dopo la notifica ne sono letti solo 1000 (ed è quindi possibile eseguire una ulteriore lettura dei restanti 1000), in modalità *level triggered* questo sarà nuovamente notificato come *pronto*.

La seconda modalità, è detta *edge triggered*, e prevede che invece vengano notificati solo i file descriptor che hanno subito una transizione da *non pronti* a *pronti*. Questo significa che in modalità *edge triggered* nel caso del precedente esempio il file descriptor diventato pronto da cui si sono letti solo 1000 byte non verrà nuovamente notificato come pronto, nonostante siano ancora disponibili in lettura 1000 byte. Solo una volta che si saranno esauriti tutti i dati disponibili, e che il file descriptor sia tornato non essere pronto, si potrà ricevere una ulteriore notifica qualora ritornasse pronto.

Nel caso di Linux al momento la sola interfaccia che fornisce questo tipo di servizio è chiamata *epoll*,²¹ anche se sono state in discussione altre interfacce con le quali effettuare lo stesso tipo di operazioni; *epoll* è in grado di operare sia in modalità *level triggered* che *edge triggered*.

La prima versione di *epoll* prevedeva l'uso di uno speciale file di dispositivo, `/dev/epoll`, per ottenere un file descriptor da utilizzare con le funzioni dell'interfaccia ma poi si è passati all'uso di apposite *system call*. Il primo passo per usare l'interfaccia di *epoll* è pertanto quello ottenere detto file descriptor chiamando una delle due funzioni di sistema `epoll_create` e `epoll_create1`, i cui prototipi sono:

```
#include <sys/epoll.h>
int epoll_create(int size)
int epoll_create1(int flags)
```

Apri un file descriptor per *epoll*.

²⁰la nomenclatura è stata introdotta da Jonathan Lemon in un articolo su `kqueue` al BSDCON 2000, e deriva da quella usata nell'elettronica digitale.

²¹l'interfaccia è stata creata da Davide Libenzi, ed è stata introdotta per la prima volta nel kernel 2.5.44, ma la sua forma definitiva è stata raggiunta nel kernel 2.5.66, il supporto è stato aggiunto nella *glibc* a partire dalla versione 2.3.2.

Le funzioni ritornano un file descriptor per *epoll* in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EINVAL	si è specificato un valore di <i>size</i> non positivo o non valido per <i>flags</i> .
EMFILE	si è raggiunto il limite sul numero massimo di istanze di <i>epoll</i> per utente stabilito da <code>/proc/sys/fs/epoll/max_user_instances</code> .
ENFILE	si è raggiunto il massimo di file descriptor aperti nel sistema.
ENOMEM	non c'è sufficiente memoria nel kernel per creare l'istanza.

Entrambe le funzioni restituiscono un file descriptor, detto anche *epoll descriptor*; si tratta di un file descriptor speciale (per cui *read* e *write* non sono supportate) che viene associato alla infrastruttura utilizzata dal kernel per gestire la notifica degli eventi, e che può a sua volta essere messo sotto osservazione con una chiamata a *select*, *poll* o *epoll_ctl*; in tal caso risulterà pronto quando saranno disponibili eventi da notificare riguardo i file descriptor da lui osservati.²² Una volta che se ne sia terminato l'uso si potranno rilasciare tutte le risorse allocate chiudendolo semplicemente con *close*.

Nel caso di *epoll_create* l'argomento *size* serviva a dare l'indicazione del numero di file descriptor che si vorranno tenere sotto controllo, e costituiva solo un suggerimento per semplificare l'allocazione di risorse sufficienti, non un valore massimo, ma a partire dal kernel 2.6.8 esso viene totalmente ignorato e l'allocazione è sempre dinamica.

La seconda versione della funzione, *epoll_create1* è stata introdotta come estensione della precedente (è disponibile solo a partire dal kernel 2.6.27) per poter passare dei flag di controllo come maschera binaria in fase di creazione del file descriptor. Al momento l'unico valore legale per *flags* (a parte lo zero) è *EPOLL_CLOEXEC*, che consente di impostare in maniera atomica sul file descriptor il flag di *close-on-exec* (vedi sez. 3.1.6 e sez. 5.2.1) senza che sia necessaria una successiva chiamata a *fcntl*.

Una volta ottenuto un file descriptor per *epoll* il passo successivo è indicare quali file descriptor mettere sotto osservazione e quali operazioni controllare, per questo si deve usare la seconda funzione di sistema dell'interfaccia, *epoll_ctl*, il cui prototipo è:

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
                Esegue le operazioni di controllo di epoll.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EBADF	i file descriptor <i>epfd</i> o <i>fd</i> non sono validi.
EEXIST	l'operazione richiesta è <i>EPOLL_CTL_ADD</i> ma <i>fd</i> è già stato inserito in <i>epfd</i> .
EINVAL	il file descriptor <i>epfd</i> non è stato ottenuto con <i>epoll_create</i> , o <i>fd</i> è lo stesso <i>epfd</i> o l'operazione richiesta con <i>op</i> non è supportata.
ENOENT	l'operazione richiesta è <i>EPOLL_CTL_MOD</i> o <i>EPOLL_CTL_DEL</i> ma <i>fd</i> non è inserito in <i>epfd</i> .
ENOMEM	non c'è sufficiente memoria nel kernel gestire l'operazione richiesta.
ENOSPC	si è raggiunto il limite massimo di registrazioni per utente di file descriptor da osservare imposto da <code>/proc/sys/fs/epoll/max_user_watches</code> .
EPERM	il file associato a <i>fd</i> non supporta l'uso di <i>epoll</i> .

La funzione prende sempre come primo argomento un file descriptor di *epoll*, *epfd*, che indica quale istanza di *epoll* usare e deve pertanto essere stato ottenuto in precedenza con una chiamata a *epoll_create* o *epoll_create1*. L'argomento *fd* indica invece il file descriptor che

²²è anche possibile inviarlo ad un altro processo attraverso un socket locale (vedi sez. 17.2.1) ma l'operazione non ha alcun senso dato che il nuovo processo non avrà a disposizione le copie dei file descriptor messe sotto osservazione tramite esso.

si vuole tenere sotto controllo, quest'ultimo può essere un qualunque file descriptor utilizzabile con `poll`, ed anche un altro file descriptor di `epoll`, ma non lo stesso `epfd`.

Il comportamento della funzione viene controllato dal valore dell'argomento `op` che consente di specificare quale operazione deve essere eseguita. Le costanti che definiscono i valori utilizzabili per `op` sono riportate in tab. 10.5, assieme al significato delle operazioni cui fanno riferimento.

Valore	Significato
EPOLL_CTL_ADD	Aggiunge un nuovo file descriptor da osservare <code>fd</code> alla lista dei file descriptor controllati tramite <code>epfd</code> , in event devono essere specificate le modalità di osservazione.
EPOLL_CTL_MOD	Modifica le modalità di osservazione del file descriptor <code>fd</code> secondo il contenuto di event.
EPOLL_CTL_DEL	Rimuove il file descriptor <code>fd</code> dalla lista dei file controllati tramite <code>epfd</code> .

Tabella 10.5: Valori dell'argomento `op` che consentono di scegliere quale operazione di controllo effettuare con la funzione `epoll_ctl`.

Le modalità di utilizzo di `epoll` prevedono che si definisca qual'è l'insieme dei file descriptor da tenere sotto controllo utilizzando una serie di chiamate a `EPOLL_CTL_ADD`.²³ L'uso di `EPOLL_CTL_MOD` consente in seguito di modificare le modalità di osservazione di un file descriptor che sia già stato aggiunto alla lista di osservazione. Qualora non si abbia più interesse nell'osservazione di un file descriptor lo si può rimuovere dalla lista associata a `epfd` con `EPOLL_CTL_DEL`.

Anche se è possibile tenere sotto controllo lo stesso file descriptor in due istanze distinte di `epoll` in genere questo è sconsigliato in quanto entrambe riceveranno le notifiche, e gestire correttamente le notifiche multiple richiede molta attenzione. Se invece si cerca di inserire due volte lo stesso file descriptor nella stessa istanza di `epoll` la funzione fallirà con un errore di `EEXIST`. Tuttavia è possibile inserire nella stessa istanza file descriptor duplicati (si ricordi quanto visto in sez. 5.2.2), una tecnica che può essere usata per registrarli con un valore diverso per `events` e classificare così diversi tipi di eventi.

Si tenga presente che quando si chiude un file descriptor questo, se era stato posto sotto osservazione da una istanza di `epoll`, viene rimosso automaticamente solo nel caso esso sia l'unico riferimento al file aperto sottostante (più precisamente alla struttura `file`, si ricordi fig. 5.4) e non è necessario usare `EPOLL_CTL_DEL`. Questo non avviene qualora esso sia stato duplicato (perché la suddetta struttura non viene disallocata) e si potranno ricevere eventi ad esso relativi anche dopo che lo si è chiuso; per evitare l'inconveniente è necessario rimuoverlo esplicitamente con `EPOLL_CTL_DEL`.

L'ultimo argomento, `event`, deve essere un puntatore ad una struttura di tipo `epoll_event`, ed ha significato solo con le operazioni `EPOLL_CTL_MOD` e `EPOLL_CTL_ADD`, per le quali serve ad indicare quale tipo di evento relativo ad `fd` si vuole che sia tenuto sotto controllo. L'argomento viene ignorato con l'operazione `EPOLL_CTL_DEL`.²⁴

La struttura `epoll_event` è l'analoga di `pollfd` e come quest'ultima serve sia in ingresso (quando usata con `epoll_ctl`) ad impostare quali eventi osservare, che in uscita (nei risultati

²³un difetto dell'interfaccia è che queste chiamate devono essere ripetute per ciascun file descriptor, incorrendo in una perdita di prestazioni qualora il numero di file descriptor sia molto grande; per questo è stato proposto di introdurre come estensione una funzione `epoll_ctlv` che consenta di effettuare con una sola chiamata le impostazioni per un blocco di file descriptor.

²⁴fino al kernel 2.6.9 era comunque richiesto che questo fosse un puntatore valido, anche se poi veniva ignorato; a partire dal 2.6.9 si può specificare anche un valore `NULL` ma se si vuole mantenere la compatibilità con le versioni precedenti occorre usare un puntatore valido.

```

typedef union epoll_data {
    void *ptr;
    int fd;
    __uint32_t u32;
    __uint64_t u64;
} epoll_data_t;

struct epoll_event {
    __uint32_t events;      /* Epoll events */
    epoll_data_t data;      /* User data variable */
};

```

Figura 10.8: La struttura `epoll_event`, che consente di specificare gli eventi associati ad un file descriptor controllato con `epoll`.

ottenuti con `epoll_wait`) per ricevere le notifiche degli eventi avvenuti. La sua definizione è riportata in fig. 10.8.

Il primo campo, `events`, è una maschera binaria in cui ciascun bit corrisponde o ad un tipo di evento, o una modalità di notifica; detto campo deve essere specificato come OR aritmetico delle costanti riportate in tab. 10.6. Nella prima parte della tabella si sono indicate le costanti che permettono di indicare il tipo di evento, che sono le equivalenti delle analoghe di tab. 10.4 per `poll`. Queste sono anche quelle riportate nella struttura `epoll_event` restituita da `epoll_wait` per indicare il tipo di evento presentatosi, insieme a quelle della seconda parte della tabella, che vengono comunque riportate anche se non le si sono impostate con `epoll_ctl`. La terza parte della tabella contiene le costanti che modificano le modalità di notifica.

Valore	Significato
EPOLLIN	Il file è pronto per le operazioni di lettura (analogo di POLLIN).
EPOLLOUT	Il file è pronto per le operazioni di scrittura (analogo di POLLOUT).
EPOLLRDHUP	L'altro capo di un socket di tipo SOCK_STREAM (vedi sez. 14.1.4) ha chiuso la connessione o il capo in scrittura della stessa (vedi sez. 15.6.3). ²⁵
EPOLLPRI	Ci sono dati urgenti disponibili in lettura (analogo di POLLPRI); questa condizione viene comunque riportata in uscita, e non è necessaria impostarla in ingresso.
EPOLLERR	Si è verificata una condizione di errore (analogo di POLLERR); questa condizione viene comunque riportata in uscita, e non è necessaria impostarla in ingresso.
EPOLLHUP	Si è verificata una condizione di hung-up; questa condizione viene comunque riportata in uscita, e non è necessaria impostarla in ingresso.
EPOLLET	Imposta la notifica in modalità <i>edge triggered</i> per il file descriptor associato.
EPOLLONESHOT	Imposta la modalità <i>one-shot</i> per il file descriptor associato (questa modalità è disponibile solo a partire dal kernel 2.6.2).
EPOLLWAKEUP	Attiva la prevenzione della sospensione del sistema se il file descriptor che si è marcato con esso diventa pronto (aggiunto a partire dal kernel 3.5), può essere impostato solo dall'amministratore (o da un processo con la capacità CAP_BLOCK_SUSPEND).

Tabella 10.6: Costanti che identificano i bit del campo `events` di `epoll_event`.

Il secondo campo, `data`, è una `union` che serve a identificare il file descriptor a cui si intende fare riferimento, ed in astratto può contenere un valore qualsiasi (specificabile in diverse forme)

²⁵questa modalità è disponibile solo a partire dal kernel 2.6.17, ed è utile per riconoscere la chiusura di una connessione dall'altro capo di un socket quando si lavora in modalità *edge triggered*.

che ne permetta una indicazione univoca. Il modo più comune di usarlo però è quello in cui si specifica il terzo argomento di `epoll_ctl` nella forma `event.data.fd`, assegnando come valore di questo campo lo stesso valore dell'argomento `fd`, cosa che permette una immediata identificazione del file descriptor.

Le impostazioni di default prevedono che la notifica degli eventi richiesti sia effettuata in modalità *level triggered*, a meno che sul file descriptor non si sia impostata la modalità *edge triggered*, registrandolo con `EPOLLET` attivo nel campo `events`.

Infine una particolare modalità di notifica è quella impostata con `EPOLLONESHOT`: a causa dell'implementazione di *epoll* infatti quando si è in modalità *edge triggered* l'arrivo in rapida successione di dati in blocchi separati (questo è tipico con i socket di rete, in quanto i dati arrivano a pacchetti) può causare una generazione di eventi (ad esempio segnalazioni di dati in lettura disponibili) anche se la condizione è già stata rilevata (si avrebbe cioè una rottura della logica *edge triggered*).

Anche se la situazione è facile da gestire, la si può evitare utilizzando `EPOLLONESHOT` per impostare la modalità *one-shot*, in cui la notifica di un evento viene effettuata una sola volta, dopo di che il file descriptor osservato, pur restando nella lista di osservazione, viene automaticamente disattivato (la cosa avviene contestualmente al ritorno di `epoll_wait` a causa dell'evento in questione) e per essere riutilizzato dovrà essere riabilitato esplicitamente con una successiva chiamata con `EPOLL_CTL_MOD`.

Una volta impostato l'insieme di file descriptor che si vogliono osservare con i relativi eventi, la funzione di sistema che consente di attendere l'occorrenza di uno di tali eventi è `epoll_wait`, il cui prototipo è:

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)
    Attende che uno dei file descriptor osservati sia pronto.
```

La funzione ritorna il numero di file descriptor pronti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` il file descriptor `epfd` non è valido.

`EFAULT` il puntatore `events` non è valido.

`EINTR` la funzione è stata interrotta da un segnale prima della scadenza di `timeout`.

`EINVAL` il file descriptor `epfd` non è stato ottenuto con `epoll_create`, o `maxevents` non è maggiore di zero.

La funzione si blocca in attesa di un evento per i file descriptor registrati nella lista di osservazione di `epfd` fino ad un tempo massimo specificato in millisecondi tramite l'argomento `timeout`. Gli eventi registrati vengono riportati in un vettore di strutture `epoll_event` (che deve essere stato allocato in precedenza) all'indirizzo indicato dall'argomento `events`, fino ad un numero massimo di eventi impostato con l'argomento `maxevents`.

La funzione ritorna il numero di eventi rilevati, o un valore nullo qualora sia scaduto il tempo massimo impostato con `timeout`. Per quest'ultimo, oltre ad un numero di millisecondi, si può utilizzare il valore nullo, che indica di non attendere e ritornare immediatamente (anche in questo caso il valore di ritorno sarà nullo) o il valore `-1`, che indica un'attesa indefinita. L'argomento `maxevents` dovrà invece essere sempre un intero positivo.

Come accennato la funzione restituisce i suoi risultati nel vettore di strutture `epoll_event` puntato da `events`; in tal caso nel campo `events` di ciascuna di esse saranno attivi i flag relativi agli eventi accaduti, mentre nel campo `data` sarà restituito il valore che era stato impostato per il file descriptor per cui si è verificato l'evento quando questo era stato registrato con le operazioni `EPOLL_CTL_MOD` o `EPOLL_CTL_ADD`, in questo modo il campo `data` consente di

identificare il file descriptor, ed è per questo che, come accennato, è consuetudine usare per data il valore del file descriptor stesso.

Si ricordi che le occasioni per cui `epoll_wait` ritorna dipendono da come si è impostata la modalità di osservazione (se *level triggered* o *edge triggered*) del singolo file descriptor. L'interfaccia assicura che se arrivano più eventi fra due chiamate successive ad `epoll_wait` questi vengano combinati. Inoltre qualora su un file descriptor fossero presenti eventi non ancora notificati, e si effettuasse una modifica dell'osservazione con `EPOLL_CTL_MOD`, questi verrebbero rilette alla luce delle modifiche.

Si tenga presente infine che con l'uso della modalità *edge triggered* il ritorno di `epoll_wait` avviene solo quando il file descriptor ha cambiato stato diventando pronto. Esso non sarà riportato nuovamente fino ad un altro cambiamento di stato, per cui occorre assicurarsi di aver completamente esaurito le operazioni su di esso. Questa condizione viene generalmente rilevata dall'occorrere di un errore di `EAGAIN` al ritorno di una `read` o una `write`, (è opportuno ricordare ancora una volta che l'uso dell'*I/O multiplexing* richiede di operare sui file in modalità non bloccante) ma questa non è la sola modalità possibile, ad esempio la condizione può essere riconosciuta anche per il fatto che sono stati restituiti meno dati di quelli richiesti.

Si tenga presente che in modalità *edge triggered*, dovendo esaurire le attività di I/O dei file descriptor risultati pronti per poter essere rinotificati, la gestione elementare per cui li si trattano uno per uno in sequenza può portare ad un effetto denominato *starvation* ("carestia"). Si rischia cioè di concentrare le operazioni sul primo file descriptor che dispone di molti dati, prolungandole per tempi molto lunghi con un ritardo che può risultare eccessivo nei confronti di quelle da eseguire sugli altri che verrebbero dopo. Per evitare questo tipo di problematiche viene consigliato di usare `epoll_wait` per registrare un elenco dei file descriptor da gestire, e di trattarli a turno in maniera più equa.

Come già per `select` e `poll` anche per l'interfaccia di *epoll* si pone il problema di gestire l'attesa di segnali e di dati contemporaneamente. Valgono le osservazioni fatte in sez. 10.2.2, e per poterlo fare di nuovo è necessaria una variante della funzione di attesa che consenta di reimpostare all'uscita una maschera di segnali, analoga alle estensioni `pselect` e `ppoll` che abbiamo visto in precedenza per `select` e `poll`. In questo caso la funzione di sistema si chiama `epoll_pwait`²⁶ ed il suo prototipo è:

```
#include <sys/epoll.h>
int epoll_pwait(int epfd, struct epoll_event * events, int maxevents, int timeout,
               const sigset_t *sigmask)
    Attende che uno dei file descriptor osservati sia pronto, mascherando i segnali.
```

La funzione ritorna il numero di file descriptor pronti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori già visti con `epoll_wait`.

La funzione è del tutto analoga `epoll_wait`, soltanto che alla sua uscita viene ripristinata la maschera di segnali originale, sostituita durante l'esecuzione da quella impostata con l'argomento `sigmask`; in sostanza la chiamata a questa funzione è equivalente al seguente codice, eseguito però in maniera atomica:

```
sigset_t origmask;
sigprocmask(SIG_SETMASK, &sigmask, &origmask);
ready = epoll_wait(epfd, &events, maxevents, timeout);
sigprocmask(SIG_SETMASK, &origmask, NULL);
```

Si tenga presente che come le precedenti funzioni di *I/O multiplexing* anche le funzioni dell'interfaccia di *epoll* vengono utilizzate prevalentemente con i server di rete, quando si

²⁶la funzione è stata introdotta a partire dal kernel 2.6.19, ed è, come tutta l'interfaccia di *epoll*, specifica di Linux.

devono tenere sotto osservazione un gran numero di socket; per questo motivo rimandiamo anche in questo caso la trattazione di un esempio concreto a quando avremo esaminato in dettaglio le caratteristiche dei socket; in particolare si potrà trovare un programma che utilizza questa interfaccia in sez. ??.

10.2.5 La notifica di eventi tramite file descriptor

Abbiamo visto in sez. 10.2.2 come il meccanismo classico delle notifiche di eventi tramite i segnali, presente da sempre nei sistemi unix-like, porti a notevoli problemi nell'interazione con le funzioni per l'I/O *multiplexing*, tanto che per evitare possibili *race condition* sono state introdotte estensioni dello standard POSIX e funzioni apposite come `pselect`, `ppoll` e `epoll_wait`.

Benché i segnali siano il meccanismo più usato per effettuare notifiche ai processi, la loro interfaccia di programmazione, che comporta l'esecuzione di una funzione di gestione in maniera asincrona e totalmente scorrelata dall'ordinario flusso di esecuzione del processo, si è però dimostrata quasi subito assai problematica. Oltre ai limiti relativi ai limiti al cosa si può fare all'interno della funzione del gestore di segnali (quelli illustrati in sez. 7.4.5), c'è il problema più generale consistente nel fatto che questa modalità di funzionamento cozza con altre interfacce di programmazione previste dal sistema in cui si opera in maniera *sincrona*, come quelle dell'I/O *multiplexing* appena illustrate.

In questo tipo di interfacce infatti ci si aspetta che il processo gestisca gli eventi a cui deve reagire in maniera sincrona generando le opportune risposte, mentre con l'arrivo di un segnale si possono avere interruzioni asincrone in qualunque momento. Questo comporta la necessità di dover gestire, quando si deve tener conto di entrambi i tipi di eventi, le interruzioni delle funzioni di attesa sincrone, ed evitare possibili *race conditions*. In sostanza se non ci fossero i segnali non ci sarebbe da preoccuparsi, fintanto che si effettuano operazioni all'interno di un processo, della non atomicità delle *system call* lente che vengono interrotte e devono essere riavviate.

Abbiamo visto però in sez. 7.5.1 che insieme ai segnali *real-time* sono state introdotte anche delle interfacce di gestione sincrona dei segnali, con la funzione `sigwait` e le sue affini. Queste funzioni consentono di gestire i segnali bloccando un processo fino alla avvenuta ricezione e disabilitando l'esecuzione asincrona rispetto al resto del programma del gestore del segnale. Questo consente di risolvere i problemi di atomicità nella gestione degli eventi associati ai segnali, avendo tutto il controllo nel flusso principale del programma, ottenendo così una gestione simile a quella dell'I/O *multiplexing*, ma non risolve i problemi delle interazioni con quest'ultimo, perché o si aspetta la ricezione di un segnale o si aspetta che un file descriptor sia accessibile e nessuna delle rispettive funzioni consente di fare contemporaneamente entrambe le cose.

Per risolvere questo problema nello sviluppo del kernel si è pensato di introdurre un meccanismo alternativo per la notifica dei segnali (esteso anche ad altri eventi generici) che, ispirandosi di nuovo alla filosofia di Unix per cui tutto è un file, consentisse di eseguire la notifica con l'uso di opportuni file descriptor. Ovviamente si tratta di una funzionalità specifica di Linux, non presente in altri sistemi unix-like, e non prevista da nessuno standard, per cui va evitata se si ha a cuore la portabilità.

In sostanza, come per `sigwait`, si può disabilitare l'esecuzione di un gestore in occasione dell'arrivo di un segnale, e rilevarne l'avvenuta ricezione leggendone la notifica tramite l'uso di uno speciale file descriptor. Trattandosi di un file descriptor questo potrà essere tenuto sotto osservazione con le ordinarie funzioni dell'I/O *multiplexing* (vale a dire con le solite `select`, `poll` e `epoll_wait`) allo stesso modo di quelli associati a file o socket, per cui alla fine

si potrà attendere in contemporanea sia l'arrivo del segnale che la disponibilità di accesso ai dati relativi a questi ultimi.

La funzione di sistema che permette di abilitare la ricezione dei segnali tramite file descriptor è `signalfd`,²⁷ il cui prototipo è:

```
#include <sys/signalfd.h>
int signalfd(int fd, const sigset_t *mask, int flags)
    Crea o modifica un file descriptor per la ricezione dei segnali.
```

La funzione ritorna un numero di file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	il valore <code>fd</code> non indica un file descriptor.
<code>EINVAL</code>	il file descriptor <code>fd</code> non è stato ottenuto con <code>signalfd</code> o il valore di <code>flags</code> non è valido.
<code>ENODEV</code>	il kernel non può montare internamente il dispositivo per la gestione anonima degli <i>inode</i> associati al file descriptor.
<code>ENOMEM</code>	non c'è memoria sufficiente per creare un nuovo file descriptor di <code>signalfd</code> .

ed inoltre `EMFILE` e `ENFILE` nel loro significato generico.

La funzione consente di creare o modificare le caratteristiche di un file descriptor speciale su cui ricevere le notifiche della ricezione di segnali. Per creare ex-novo uno di questi file descriptor è necessario passare `-1` come valore per l'argomento `fd`, ogni altro valore positivo verrà invece interpretato come il numero del file descriptor (che deve esser stato precedentemente creato sempre con `signalfd`) di cui si vogliono modificare le caratteristiche. Nel primo caso la funzione ritornerà il valore del nuovo file descriptor e nel secondo caso il valore indicato con `fd`, in caso di errore invece verrà restituito `-1`.

L'elenco dei segnali che si vogliono gestire con `signalfd` deve essere specificato tramite l'argomento `mask`. Questo deve essere passato come puntatore ad una maschera di segnali creata con l'uso delle apposite macro già illustrate in sez. 7.4.2. La maschera deve indicare su quali segnali si intende operare con `signalfd`; l'elenco può essere modificato con una successiva chiamata a `signalfd`. Dato che `SIGKILL` e `SIGSTOP` non possono essere intercettati (e non prevedono neanche la possibilità di un gestore) un loro inserimento nella maschera verrà ignorato senza generare errori.

L'argomento `flags` consente di impostare direttamente in fase di creazione due flag per il file descriptor analoghi a quelli che si possono impostare con una creazione ordinaria con `open`, evitando una impostazione successiva con `fcntl` (si ricordi che questo è un argomento aggiuntivo, introdotto con la versione fornita a partire dal kernel 2.6.27, per kernel precedenti il valore deve essere nullo). L'argomento deve essere specificato come maschera binaria dei valori riportati in tab. 10.7.

Valore	Significato
<code>SFD_NONBLOCK</code>	imposta sul file descriptor il flag di <code>O_NONBLOCK</code> per renderlo non bloccante.
<code>SFD_CLOEXEC</code>	imposta il flag di <code>O_CLOEXEC</code> per la chiusura automatica del file descriptor nella esecuzione di <code>exec</code> .

Tabella 10.7: Valori dell'argomento `flags` per la funzione `signalfd` che consentono di impostare i flag del file descriptor.

²⁷in realtà quella riportata è l'interfaccia alla funzione fornita dalla *glibc*, esistono infatti due versioni diverse della *system call*; una prima versione, `signalfd`, introdotta nel kernel 2.6.22 e disponibile con la *glibc* 2.8 che non supporta l'argomento `flags`, ed una seconda versione, `signalfd4`, introdotta con il kernel 2.6.27 e che è quella che viene sempre usata a partire dalla *glibc* 2.9, che prende un argomento aggiuntivo `size_t sizemask` che indica la dimensione della maschera dei segnali, il cui valore viene impostato automaticamente dalla *glibc*.

Si tenga presente che la chiamata a `signalfd` non disabilita la gestione ordinaria dei segnali indicati da `mask`; questa, se si vuole effettuare la ricezione tramite il file descriptor, dovrà essere disabilitata esplicitamente bloccando gli stessi segnali con `sigprocmask`, altrimenti verranno comunque eseguite le azioni di default (o un eventuale gestore installato in precedenza). Il blocco non ha invece nessun effetto sul file descriptor restituito da `signalfd`, dal quale sarà possibile pertanto ricevere qualunque segnale, anche se questo risultasse bloccato.

Si tenga presente inoltre che la lettura di una struttura `signalfd_siginfo` relativa ad un segnale pendente è equivalente alla esecuzione di un gestore, vale a dire che una volta letta il segnale non sarà più pendente e non potrà essere ricevuto, qualora si ripristino le normali condizioni di gestione, né da un gestore, né dalla funzione `sigwaitinfo`.

Come anticipato, essendo questo lo scopo principale della nuova interfaccia, il file descriptor può essere tenuto sotto osservazione tramite le funzioni dell'*I/O multiplexing* (vale a dire con le solite `select`, `poll` e `epoll_wait`), e risulterà accessibile in lettura quando uno o più dei segnali indicati tramite `mask` sarà pendente.

La funzione può essere chiamata più volte dallo stesso processo, consentendo così di tenere sotto osservazione segnali diversi tramite file descriptor diversi. Inoltre è anche possibile tenere sotto osservazione lo stesso segnale con più file descriptor, anche se la pratica è sconsigliata; in tal caso la ricezione del segnale potrà essere effettuata con una lettura da uno qualunque dei file descriptor a cui è associato, ma questa potrà essere eseguita soltanto una volta. Questo significa che tutti i file descriptor su cui è presente lo stesso segnale risulteranno pronti in lettura per le funzioni di *I/O multiplexing*, ma una volta eseguita la lettura su uno di essi il segnale sarà considerato ricevuto ed i relativi dati non saranno più disponibili sugli altri file descriptor, che (a meno di una ulteriore occorrenza del segnale nel frattempo) di non saranno più pronti.

Quando il file descriptor per la ricezione dei segnali non serve più potrà essere chiuso con `close` liberando tutte le risorse da esso allocate. In tal caso qualora vi fossero segnali pendenti questi resteranno tali, e potranno essere ricevuti normalmente una volta che si rimuova il blocco imposto con `sigprocmask`.

Oltre a poter essere usato con le funzioni dell'*I/O multiplexing*, il file descriptor restituito da `signalfd` cerca di seguire la semantica di un sistema unix-like anche con altre *system call*; in particolare esso resta aperto (come ogni altro file descriptor) attraverso una chiamata ad `exec`, a meno che non lo si sia creato con il flag di `SFD_CLOEXEC` o si sia successivamente impostato il *close-on-exec* con `fcntl`. Questo comportamento corrisponde anche alla ordinaria semantica relativa ai segnali bloccati, che restano pendenti attraverso una `exec`.

Analogamente il file descriptor resta sempre disponibile attraverso una `fork` per il processo figlio, che ne riceve una copia; in tal caso però il figlio potrà leggere dallo stesso soltanto i dati relativi ai segnali ricevuti da lui stesso. Nel caso di *thread* viene nuovamente seguita la semantica ordinaria dei segnali, che prevede che un singolo *thread* possa ricevere dal file descriptor solo le notifiche di segnali inviati direttamente a lui o al processo in generale, e non quelli relativi ad altri *thread* appartenenti allo stesso processo.

L'interfaccia fornita da `signalfd` prevede che la ricezione dei segnali sia eseguita leggendo i dati relativi ai segnali pendenti dal file descriptor restituito dalla funzione con una normalissima `read`. Qualora non vi siano segnali pendenti la `read` si bloccherà a meno di non aver impostato la modalità di I/O non bloccante sul file descriptor, o direttamente in fase di creazione con il flag `SFD_NONBLOCK`, o in un momento successivo con `fcntl`.

I dati letti dal file descriptor vengono scritti sul buffer indicato come secondo argomento di `read` nella forma di una sequenza di una o più strutture `signalfd_siginfo` (la cui definizione si è riportata in fig. 10.9) a seconda sia della dimensione del buffer che del numero di segnali pendenti. Per questo motivo il buffer deve essere almeno di dimensione pari a quella

```

struct signalfd_siginfo {
    uint32_t ssi_signo; /* Signal number */
    int32_t ssi_errno; /* Error number (unused) */
    int32_t ssi_code; /* Signal code */
    uint32_t ssi_pid; /* PID of sender */
    uint32_t ssi_uid; /* Real UID of sender */
    int32_t ssi_fd; /* File descriptor (SIGIO) */
    uint32_t ssi_tid; /* Kernel timer ID (POSIX timers) */
    uint32_t ssi_band; /* Band event (SIGIO) */
    uint32_t ssi_overrun; /* POSIX timer overrun count */
    uint32_t ssi_trapno; /* Trap number that caused signal */
    int32_t ssi_status; /* Exit status or signal (SIGCHLD) */
    int32_t ssi_int; /* Integer sent by sigqueue(2) */
    uint64_t ssi_ptr; /* Pointer sent by sigqueue(2) */
    uint64_t ssi_utime; /* User CPU time consumed (SIGCHLD) */
    uint64_t ssi_stime; /* System CPU time consumed (SIGCHLD) */
    uint64_t ssi_addr; /* Address that generated signal
                        (for hardware-generated signals) */
    uint8_t pad[X]; /* Pad size to 128 bytes (allow for
                    additional fields in the future) */
};

```

Figura 10.9: La struttura `signalfd_siginfo`, restituita in lettura da un file descriptor creato con `signalfd`.

di `signalfd_siginfo`, qualora sia di dimensione maggiore potranno essere letti in unica soluzione i dati relativi ad eventuali più segnali pendenti, fino al numero massimo di strutture `signalfd_siginfo` che possono rientrare nel buffer.

Il contenuto di `signalfd_siginfo` ricalca da vicino quella dell'analoga struttura `siginfo_t` (illustrata in fig. 7.9) usata dall'interfaccia ordinaria dei segnali, e restituisce dati simili. Come per `siginfo_t` i campi che vengono avvalorati dipendono dal tipo di segnale e ricalcano i valori che abbiamo già illustrato in sez. 7.4.3.²⁸

Come esempio di questa nuova interfaccia ed anche come esempio di applicazione della interfaccia di *epoll*, si è scritto un programma elementare che stampi sullo *standard output* sia quanto viene scritto da terzi su una *named fifo*, che l'avvenuta ricezione di alcuni segnali. Il codice completo si trova al solito nei sorgenti allegati alla guida (nel file `FifoReporter.c`).

In fig. 10.10 si è riportata la parte iniziale del programma in cui vengono effettuate le varie inizializzazioni necessarie per l'uso di *epoll* e `signalfd`, a partire (12-16) dalla definizione delle varie variabili e strutture necessarie. Al solito si è tralasciata la parte dedicata alla decodifica delle opzioni che consentono ad esempio di cambiare il nome del file associato alla *fifo*.

Il primo passo (19-20) è la creazione di un file descriptor `epfd` di *epoll* con `epoll_create` che è quello che useremo per il controllo degli altri. È poi necessario disabilitare la ricezione dei segnali (nel caso `SIGINT`, `SIGQUIT` e `SIGTERM`) per i quali si vuole la notifica tramite file descriptor. Per questo prima li si inseriscono (22-25) in una maschera di segnali `sigmask` che useremo con (26) `sigprocmask` per disabilitarli. Con la stessa maschera si potrà per passare all'uso (28-29) di `signalfd` per abilitare la notifica sul file descriptor `sigfd`. Questo poi (30-33) dovrà essere aggiunto con `epoll_ctl` all'elenco di file descriptor controllati con `epfd`.

Occorrerà infine (35-38) creare la *named fifo* se questa non esiste ed aprirla per la lettura (39-40); una volta fatto questo sarà necessario aggiungere il relativo file descriptor (`fifofd`)

²⁸si tenga presente però che per un bug i kernel fino al 2.6.25 non avvalorano correttamente i campi `ssi_ptr` e `ssi_int` per segnali inviati con `sigqueue`.

```

1 ...
2 #include <sys/epoll.h>      /* Linux epoll interface */
3 #include <sys/signalfd.h>   /* Linux signalfd interface */
4
5 void die(char *);           /* print error and exit function */
6 #define MAX_EPOLL_EV 10
7 int main(int argc, char *argv[])
8 {
9     /* Variables definition */
10     int i, n, nread, t = 10;
11     char buffer[4096];
12     int fifofd, epfd, sigfd;
13     sigset_t sigmask;
14     char *fifoname = "/tmp/reporter.fifo";
15     struct epoll_event epev, events[MAX_EPOLL_EV];
16     struct signalfd_siginfo siginf;
17     ...
18     /* Initial setup */
19     if ((epfd=epoll_create(5)) < 0)                                // epoll init
20         die("Failing_on_epoll_create");
21     /* Signal setup for signalfd and epoll use */
22     sigemptyset(&sigmask);
23     sigaddset(&sigmask, SIGINT);
24     sigaddset(&sigmask, SIGQUIT);
25     sigaddset(&sigmask, SIGTERM);
26     if (sigprocmask(SIG_BLOCK, &sigmask, NULL) == -1)            // block signals
27         die("Failing_in_sigprocmask");
28     if ((sigfd=signalfd(-1, &sigmask, SFD_NONBLOCK)) == -1) // take a signalfd
29         die("Failing_in_signalfd");
30     epev.data.fd = sigfd;                                          // add fd to epoll
31     epev.events = EPOLLIN;
32     if (epoll_ctl(epfd, EPOLL_CTL_ADD, sigfd, &epev))
33         die("Failing_in_signal_epoll_ctl");
34     /* Fifo setup for epoll use */
35     if (mkfifo(fifoname, 0622)) { // create well known fifo if does't exist
36         if (errno!=EEXIST)
37             die("Cannot_create_well_known_fifo");
38     }
39     if ((fifofd = open(fifoname, O_RDWR|O_NONBLOCK)) < 0) // open fifo
40         die("Cannot_open_read_only_well_known_fifo");
41     epev.data.fd = fifofd;                                          // add fd to epoll
42     epev.events = EPOLLIN;
43     if (epoll_ctl(epfd, EPOLL_CTL_ADD, fifofd, &epev))
44         die("Failing_in_fifo_epoll_ctl");
45     /* Main body: wait something to report */
46     ...
47 }

```

Figura 10.10: Sezione di inizializzazione del codice del programma `FifoReporter.c`.

a quelli osservati da *epoll* in maniera del tutto analoga a quanto fatto con quello relativo alla notifica dei segnali.

Una volta completata l'inizializzazione verrà eseguito indefinitamente il ciclo principale del programma (2-45) che si è riportato in fig. 10.11, fintanto che questo non riceva un segnale di SIGINT (ad esempio con la pressione di C-c). Il ciclo prevede che si attenda (2-3) la presenza di un file descriptor pronto in lettura con *epoll_wait* (si ricordi che entrambi i file descriptor

```

1  /* Main body: wait something to report */
2  while (1) {
3      if ((n=epoll_wait(epfd, events, MAX_EPOLL_EV, -1)) < 0)
4          die("error_on_epoll_wait");
5      for (i=0; i<n; i++) { // loop on ready file descriptors
6          if (events[i].data.fd == sigfd) { // look if signalfd ready
7              printf("Signal_received:\n");
8              while(nread=read(sigfd, &siginfo, sizeof(siginfo))) {
9                  if (nread < 0) {
10                     if (errno != EAGAIN)
11                         die("signalfd_read_error");
12                     else
13                         break;
14                 }
15                 if (nread != sizeof(siginfo)) {
16                     printf("Error_on_signal_data_read,\n");
17                     continue;
18                 }
19                 printf("Got_%s\n", sig_names[siginfo.ssi_signo]);
20                 printf("From_pid_%i\n", siginfo.ssi_pid);
21                 if(siginfo.ssi_signo == SIGINT) { // SIGINT stop program
22                     unlink(fifoname);
23                     exit(0);
24                 }
25             }
26         } else if (events[i].data.fd == fifofd) { // look if fifofd ready
27             printf("Message_from_fifo:\n");
28             while ((nread = read(fifofd, buffer, 5000))) {
29                 if (nread < 0) {
30                     if (errno != EAGAIN)
31                         die("fifo_read_error");
32                     else
33                         printf("end_message\n");
34                     break;
35                 }
36                 buffer[nread] = 0;
37                 if (fputs(buffer, stdout) == EOF)
38                     die("Errore_in_scrittura_su_terminale");
39             }
40         } else { // anything else is an error
41             printf("epoll_activity_on_unknown_%i_file_descriptor\n",
42                    epev.data.fd);
43             exit(-1);
44         }
45     }
46 }

```

Figura 10.11: Ciclo principale del codice del programma FifoReporter.c.

fifofd e sigfd sono stati posti in osservazioni per eventi di tipo EPOLLIN) che si bloccherà fintanto che non siano stati scritti dati sulla *fifo* o che non sia arrivato un segnale.²⁹

Anche se in questo caso i file descriptor pronti possono essere al più due, si è comunque adottato un approccio generico in cui questi verranno letti all'interno di un opportuno ciclo

²⁹per semplificare il codice non si è trattato il caso in cui `epoll_wait` viene interrotta da un segnale, assumendo che tutti quelli che possano interessare siano stati predisposti per la notifica tramite file descriptor, per gli altri si otterrà semplicemente l'uscita dal programma.

(5-44) sul numero restituito da `epoll_wait`, esaminando i risultati presenti nel vettore `events` all'interno di una catena di condizionali alternativi sul valore del file descriptor riconosciuto come pronto, controllando cioè a quale dei due file descriptor possibili corrisponde il campo relativo, `events[i].data.fd`.

Il primo condizionale (6-24) è relativo al caso che si sia ricevuto un segnale e che il file descriptor pronto corrisponda (6) a `sigfd`. Dato che in generale si possono ricevere anche notifiche relativi a più di un singolo segnale, si è scelto di leggere una struttura `sigalfd_siginfo` alla volta, eseguendo la lettura all'interno di un ciclo (8-24) che prosegue fintanto che vi siano dati da leggere.

Per questo ad ogni lettura si esamina (9-14) se il valore di ritorno della funzione `read` è negativo, uscendo dal programma (11) in caso di errore reale, o terminando il ciclo (13) con un `break` qualora si ottenga un errore di `EAGAIN` per via dell'esaurimento dei dati. Si ricordi infatti come sia la *fifo* che il file descriptor per i segnali siano stati aperti in modalità non-bloccante, come previsto per l'I/O *multiplexing*, pertanto ci si aspetta di ricevere un errore di `EAGAIN` quando non vi saranno più dati da leggere.

In presenza di dati invece il programma proseguirà l'esecuzione stampando (19-20) il nome del segnale ottenuto all'interno della struttura `sigalfd_siginfo` letta in `siginfo` ed il *pid* del processo da cui lo ha ricevuto;³⁰ inoltre (21-24) si controllerà anche se il segnale ricevuto è `SIGINT`, che si è preso come segnale da utilizzare per la terminazione del programma, che verrà eseguita dopo aver rimosso il file della *name fifo*.

Il secondo condizionale (26-39) è invece relativo al caso in cui ci siano dati pronti in lettura sulla *fifo* e che il file descriptor pronto corrisponda (26) a `fifo`. Di nuovo si effettueranno le letture in un ciclo (28-39) ripetendole fin tanto che la funzione `read` non restituisce un errore di `EAGAIN` (29-35). Il procedimento è lo stesso adottato per il file descriptor associato al segnale, in cui si esce dal programma in caso di errore reale, in questo caso però alla fine dei dati prima di uscire si stampa anche (32) un messaggio di chiusura.

Se invece vi sono dati validi letti dalla *fifo* si inserirà (36) una terminazione di stringa sul buffer e si stamperà il tutto (37-38) sullo *standard output*. L'ultimo condizionale (40-44) è semplicemente una condizione di cattura per una eventualità che comunque non dovrebbe mai verificarsi, e che porta alla uscita dal programma con una opportuna segnalazione di errore.

A questo punto si potrà eseguire il comando lanciandolo su un terminale, ed osservarne le reazioni agli eventi generati da un altro terminale; lanciando il programma otterremo qualcosa del tipo:

```
piccardi@hain:~/gapil/sources$ ./a.out
FifoReporter starting, pid 4568
```

e scrivendo qualcosa sull'altro terminale con:

```
root@hain:~# echo prova > /tmp/reporter.fifo
```

si otterrà:

```
Message from fifo:
prova
end message
```

mentre inviando un segnale:

```
root@hain:~# kill 4568
```

si avrà:

³⁰per la stampa si è usato il vettore `sig_names` a ciascun elemento del quale corrisponde il nome del segnale avente il numero corrispondente, la cui definizione si è omessa dal codice di fig. 10.10 per brevità.

```
Signal received:
Got SIGTERM
From pid 3361
```

ed infine premendo C-\ sul terminale in cui è in esecuzione si vedrà:

```
^                               Signal received:
Got SIGQUIT
From pid 0
```

e si potrà far uscire il programma con C-c ottenendo:

```
^CSignal received:
Got SIGINT
From pid 0
SIGINT means exit
```

Lo stesso paradigma di notifica tramite file descriptor usato per i segnali è stato adottato anche per i timer. In questo caso, rispetto a quanto visto in sez. 7.5.2, la scadenza di un timer potrà essere letta da un file descriptor senza dover ricorrere ad altri meccanismi di notifica come un segnale o un *thread*. Di nuovo questo ha il vantaggio di poter utilizzare le funzioni dell'*I/O multiplexing* per attendere allo stesso tempo la disponibilità di dati o la ricezione della scadenza di un timer. In realtà per questo sarebbe già sufficiente `signalfd` per ricevere i segnali associati ai timer, ma la nuova interfaccia semplifica notevolmente la gestione e consente di fare tutto con una sola *system call*.

Le funzioni di questa nuova interfaccia ricalcano da vicino la struttura delle analoghe versioni ordinarie introdotte con lo standard POSIX.1-2001, che abbiamo già illustrato in sez. 7.5.2.³¹ La prima funzione di sistema prevista, quella che consente di creare un timer, è `timerfd_create`, il cui prototipo è:

<pre>#include <sys/timerfd.h> int timerfd_create(int clockid, int flags)</pre> <p style="text-align: right;">Crea un timer associato ad un file descriptor di notifica.</p>
<p>La funzione ritorna un numero di file descriptor in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EINVAL l'argomento <code>clockid</code> non è <code>CLOCK_MONOTONIC</code> o <code>CLOCK_REALTIME</code>, o l'argomento <code>flag</code> non è valido, o è diverso da zero per kernel precedenti il 2.6.27.</p> <p>ENODEV il kernel non può montare internamente il dispositivo per la gestione anonima degli <i>inode</i> associati al file descriptor.</p> <p>ENOMEM non c'è memoria sufficiente per creare un nuovo file descriptor di <code>signalfd</code>. ed inoltre <code>EMFILE</code> e <code>ENFILE</code> nel loro significato generico.</p>

La funzione prende come primo argomento un intero che indica il tipo di orologio a cui il timer deve fare riferimento, i valori sono gli stessi delle funzioni dello standard POSIX-1.2001 già illustrati in tab. 7.10, ma al momento i soli utilizzabili sono `CLOCK_REALTIME` e `CLOCK_MONOTONIC`. L'argomento `flags`, come l'analogo di `signalfd`, consente di impostare i flag per l'I/O non bloccante ed il *close-on-exec* sul file descriptor restituito,³² e deve essere specificato come una maschera binaria delle costanti riportate in tab. 10.8.

In caso di successo la funzione restituisce un file descriptor sul quale verranno notificate le scadenze dei timer. Come per quelli restituiti da `signalfd` anche questo file descriptor segue

³¹questa interfaccia è stata introdotta in forma considerata difettosa con il kernel 2.6.22, per cui è stata immediatamente tolta nel successivo 2.6.23 e reintrodotta in una forma considerata adeguata nel kernel 2.6.25, il supporto nella *glibc* è stato introdotto a partire dalla versione 2.8.6, la versione del kernel 2.6.22, presente solo su questo kernel, non è supportata e non deve essere usata.

³²il flag è stato introdotto a partire dal kernel 2.6.27, per le versioni precedenti deve essere passato un valore nullo.

Valore	Significato
TFD_NONBLOCK	imposta sul file descriptor il flag di O_NONBLOCK per renderlo non bloccante.
TFD_CLOEXEC	imposta il flag di O_CLOEXEC per la chiusura automatica del file descriptor nella esecuzione di exec.

Tabella 10.8: Valori dell'argomento flags per la funzione `timerfd_create` che consentono di impostare i flag del file descriptor.

la semantica dei sistemi unix-like, in particolare resta aperto attraverso una `exec` (a meno che non si sia impostato il flag di *close-on exec* con `TFD_CLOEXEC`) e viene duplicato attraverso una `fork`; questa ultima caratteristica comporta però che anche il figlio può utilizzare i dati di un timer creato nel padre, a differenza di quanto avviene invece con i timer impostati con le funzioni ordinarie. Si ricordi infatti che, come illustrato in sez. 3.1.3, allarmi, timer e segnali pendenti nel padre vengono cancellati per il figlio dopo una `fork`.

Una volta creato il timer con `timerfd_create` per poterlo utilizzare occorre *armarlo* impostandone un tempo di scadenza ed una eventuale periodicità di ripetizione, per farlo si usa una funzione di sistema omologa di `timer_settime` per la nuova interfaccia; questa è `timerfd_settime` ed il suo prototipo è:

```
#include <sys/timerfd.h>
int timerfd_settime(int fd, int flags, const struct itimerspec *new_value,
                   struct itimerspec *old_value)
    Arma un timer associato ad un file descriptor di notifica.
```

La funzione ritorna un numero di file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF l'argomento `fd` non corrisponde ad un file descriptor.
EFAULT o `new_value` o `old_value` non sono puntatori validi.
EINVAL il file descriptor `fd` non è stato ottenuto con `timerfd_create`, o i valori di `flag` o dei campi `tv_nsec` in `new_value` non sono validi.

In questo caso occorre indicare su quale timer si intende operare specificando come primo argomento il file descriptor ad esso associato, che deve essere stato ottenuto da una precedente chiamata a `timerfd_create`. I restanti argomenti sono del tutto analoghi a quelli della omologa funzione `timer_settime`, e prevedono l'uso di strutture `itimerspec` (vedi fig. 7.16) per le indicazioni di temporizzazione.

I valori ed il significato di questi argomenti sono gli stessi che sono già stati illustrati in dettaglio in sez. 7.5.2 e non staremo a ripetere quanto detto in quell'occasione; per brevità si ricordi che con `new_value.it_value` si indica la prima scadenza del timer e con `new_value.it_interval` la sua periodicità. L'unica differenza riguarda l'argomento `flags` che serve sempre ad indicare se il tempo di scadenza del timer è da considerarsi relativo o assoluto rispetto al valore corrente dell'orologio associato al timer, ma che in questo caso ha come valori possibili rispettivamente soltanto 0 e `TFD_TIMER_ABSTIME` (l'analogo di `TIMER_ABSTIME`).

L'ultima funzione di sistema prevista dalla nuova interfaccia è `timerfd_gettime`, che è l'analogo di `timer_gettime`, il suo prototipo è:

```
#include <sys/timerfd.h>
int timerfd_gettime(int fd, struct itimerspec *curr_value)
    Legge l'impostazione di un timer associato ad un file descriptor di notifica.
```

La funzione ritorna un numero di file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	l'argomento <code>fd</code> non corrisponde ad un file descriptor.
<code>EINVAL</code>	il file descriptor <code>fd</code> non è stato ottenuto con <code>timerfd_create</code> .
<code>EFAULT</code>	o <code>curr_value</code> non è un puntatore valido.

La funzione consente di rileggere le impostazioni del timer associato al file descriptor `fd` nella struttura `itimerspec` puntata da `curr_value`. Il campo `it_value` riporta il tempo rimanente alla prossima scadenza del timer, che viene sempre espresso in forma relativa, anche se lo si è armato specificando `TFD_TIMER_ABSTIME`. Un valore nullo (di entrambi i campi di `it_value`) indica invece che il timer non è stato ancora armato. Il campo `it_interval` riporta la durata dell'intervallo di ripetizione del timer, ed un valore nullo (di entrambi i campi) indica che il timer è stato impostato per scadere una sola volta.

Il timer creato con `timerfd_create` notificherà la sua scadenza rendendo pronto per la lettura il file descriptor ad esso associato, che pertanto potrà essere messo sotto controllo con una qualunque delle varie funzioni dell'I/O multiplexing viste in precedenza. Una volta che il file descriptor risulta pronto sarà possibile leggere il numero di volte che il timer è scaduto con una ordinaria `read`.

La funzione legge il valore in un dato di tipo `uint64_t`, e necessita pertanto che le si passi un buffer di almeno 8 byte, fallendo con `EINVAL` in caso contrario, in sostanza la lettura deve essere effettuata con una istruzione del tipo:

```
uint64_t nexp;
...
read(fd, &nexp, sizeof(uint64_t));
```

Il valore viene restituito da `read` seguendo l'ordinamento dei bit (*big-endian* o *little-endian*) nativo della macchina in uso, ed indica il numero di volte che il timer è scaduto dall'ultima lettura eseguita con successo, o, se lo si legge per la prima volta, da quando lo si è impostato con `timerfd_settime`. Se il timer non è scaduto la funzione si blocca fino alla prima scadenza, a meno di non aver creato il file descriptor in modalità non bloccante con `TFD_NONBLOCK` o aver impostato la stessa con `fcntl`, nel qual caso fallisce con l'errore di `EAGAIN`.

10.3 L'accesso *asincrono* ai file

Benché l'I/O *multiplexing* sia stata la prima, e sia tutt'ora una fra le più diffuse modalità di gestire l'I/O in situazioni complesse in cui si debba operare su più file contemporaneamente, esistono altre modalità di gestione delle stesse problematiche. In particolare sono importanti in questo contesto le modalità di accesso ai file eseguibili in maniera *asincrona*, quelle cioè in cui un processo non deve bloccarsi in attesa della disponibilità dell'accesso al file, ma può proseguire nell'esecuzione utilizzando invece un meccanismo di notifica asincrono (di norma un segnale, ma esistono anche altre interfacce, come *inotify*), per essere avvisato della possibilità di eseguire le operazioni di I/O volute.

10.3.1 Il *Signal driven I/O*

Abbiamo accennato in sez. 5.1.2 che è definito un flag `O_ASYNC`, che consentirebbe di aprire un file in modalità asincrona, anche se in realtà è opportuno attivare in un secondo tempo questa modalità impostando questo flag attraverso l'uso di `fcntl` con il comando `F_SETFL` (vedi

sez. 5.2.5).³³ In realtà parlare di apertura in modalità asincrona non significa che le operazioni di lettura o scrittura del file vengono eseguite in modo asincrono (tratteremo questo, che è ciò che più propriamente viene chiamato *I/O asincrono*, in sez. 10.3.3), quanto dell'attivazione un meccanismo di notifica asincrona delle variazioni dello stato del file descriptor aperto in questo modo.

Quello che succede è che per tutti i file posti in questa modalità il sistema genera un apposito segnale, `SIGIO`, tutte le volte che diventa possibile leggere o scrivere dal file descriptor; si tenga presente però che essa non è utilizzabile con i file ordinari ma solo con socket, file di terminale o pseudo terminale, ed anche, a partire dal kernel 2.6, per *fifo* e *pipe*. Inoltre è possibile, come illustrato in sez. 5.2.5, selezionare con il comando `F_SETOWN` di `fcntl` quale processo o quale gruppo di processi dovrà ricevere il segnale. In questo modo diventa possibile effettuare le operazioni di I/O in risposta alla ricezione del segnale, e non ci sarà più la necessità di restare bloccati in attesa della disponibilità di accesso ai file.

Per questo motivo Stevens, ed anche le pagine di manuale di Linux, chiamano questa modalità "*Signal driven I/O*". Si tratta di un'altra modalità di gestione dell'I/O, alternativa all'uso di *epoll*,³⁴ che consente di evitare l'uso delle funzioni `poll` o `select` che, come illustrato in sez. 10.2.4, quando vengono usate con un numero molto grande di file descriptor, non hanno buone prestazioni.

Tuttavia con l'implementazione classica dei segnali questa modalità di I/O presenta notevoli problemi, dato che non è possibile determinare, quando i file descriptor sono più di uno, qual è quello responsabile dell'emissione del segnale. Inoltre dato che i segnali normali non si accodano (si ricordi quanto illustrato in sez. 7.1.4), in presenza di più file descriptor attivi contemporaneamente, più segnali emessi nello stesso momento verrebbero notificati una volta sola.

Linux però supporta le estensioni POSIX.1b dei segnali *real-time*, che vengono accodati e che permettono di riconoscere il file descriptor che li ha emessi. In questo caso infatti si può fare ricorso alle informazioni aggiuntive restituite attraverso la struttura `siginfo_t`, utilizzando la forma estesa `sa_sigaction` del gestore installata con il flag `SA_SIGINFO` (si riveda quanto illustrato in sez. 7.4.3).

Per far questo però occorre utilizzare le funzionalità dei segnali *real-time* (vedi sez. 7.5.1) impostando esplicitamente con il comando `F_SETSIG` di `fcntl` un segnale *real-time* da inviare in caso di I/O asincrono (il segnale predefinito è `SIGIO`). In questo caso il gestore, tutte le volte che riceverà `SI_SIGIO` come valore del campo `si_code` di `siginfo_t`, troverà nel campo `si_fd` il valore del file descriptor che ha generato il segnale. Si noti che il valore `si_code` resta `SI_SIGIO` qualunque sia il segnale che si è associato all'I/O, in quanto indica che il segnale è stato generato a causa di attività di I/O.

Un secondo vantaggio dell'uso dei segnali *real-time* è che essendo questi ultimi dotati di una coda di consegna ogni segnale sarà associato ad uno solo file descriptor; inoltre sarà possibile stabilire delle priorità nella risposta a seconda del segnale usato, dato che i segnali *real-time* supportano anche questa funzionalità. In questo modo si può identificare immediatamente un file su cui l'accesso è diventato possibile evitando completamente l'uso di funzioni come `poll` e `select`, almeno fintanto che non si satura la coda.

Se infatti si eccedono le dimensioni di quest'ultima, il kernel, non potendo più assicurare il comportamento corretto per un segnale *real-time*, invierà al suo posto un solo `SIGIO`, su cui si saranno accumulati tutti i segnali in eccesso, e si dovrà allora determinare con un

³³l'uso del flag di `O_ASYNC` e dei comandi `F_SETOWN` e `F_GETOWN` per `fcntl` è specifico di Linux e BSD.

³⁴anche se le prestazioni ottenute con questa tecnica sono inferiori, il vantaggio è che questa modalità è utilizzabile anche con kernel che non supportano *epoll*, come quelli della serie 2.4, ottenendo comunque prestazioni superiori a quelle che si hanno con `poll` e `select`.

ciclo quali sono i file diventati attivi. L'unico modo per essere sicuri che questo non avvenga è di impostare la lunghezza della coda dei segnali *real-time* ad una dimensione identica al valore massimo del numero di file descriptor utilizzabili, vale a dire impostare il contenuto di `/proc/sys/kernel/rtsig-max` allo stesso valore del contenuto di `/proc/sys/fs/file-max`.

10.3.2 I meccanismi di notifica asincrona.

Una delle domande più frequenti nella programmazione in ambiente unix-like è quella di come fare a sapere quando un file viene modificato. La risposta, o meglio la non risposta, tanto che questa nelle Unix FAQ [?] viene anche chiamata una *Frequently Unanswered Question*, è che nell'architettura classica di Unix questo non è possibile. Al contrario di altri sistemi operativi infatti un kernel unix-like classico non prevedeva alcun meccanismo per cui un processo possa essere *notificato* di eventuali modifiche avvenute su un file.

Questo è il motivo per cui i demoni devono essere *avvisati* in qualche modo se il loro file di configurazione è stato modificato, perché possano rileggerlo e riconoscere le modifiche; in genere questo vien fatto inviandogli un segnale di `SIGHUP` che, per una convenzione adottata dalla gran parte di detti programmi, causa la riletture della configurazione.

Questa scelta è stata fatta perché provvedere un simile meccanismo a livello generico per qualunque file comporterebbe un notevole aumento di complessità dell'architettura della gestione dei file, il tutto per fornire una funzionalità che serve soltanto in alcuni casi particolari. Dato che all'origine di Unix i soli programmi che potevano avere una tale esigenza erano i demoni, attenendosi a uno dei criteri base della progettazione, che era di far fare al kernel solo le operazioni strettamente necessarie e lasciare tutto il resto a processi in *user space*, non era stata prevista nessuna funzionalità di notifica.

Visto però il crescente interesse nei confronti di una funzionalità di questo tipo, che è molto richiesta specialmente nello sviluppo dei programmi ad interfaccia grafica quando si deve presentare all'utente lo stato del filesystem, sono state successivamente introdotte delle estensioni che permettessero la creazione di meccanismi di notifica più efficienti dell'unica soluzione disponibile con l'interfaccia tradizionale, che è quella del *polling*.

Queste nuove funzionalità sono delle estensioni specifiche, non standardizzate, che sono disponibili soltanto su Linux (anche se altri kernel supportano meccanismi simili). Alcune di esse sono realizzate, e solo a partire dalla versione 2.4 del kernel, attraverso l'uso di alcuni comandi aggiuntivi per la funzione `fcntl` (vedi sez. 5.2.5), che divengono disponibili soltanto se si è definita la macro `_GNU_SOURCE` prima di includere `fcntl.h`.

La prima di queste funzionalità è quella del cosiddetto *file lease*; questo è un meccanismo che consente ad un processo, detto *lease holder*, di essere notificato quando un altro processo, chiamato a sua volta *lease breaker*, cerca di eseguire una `open` o una `truncate` sul file del quale l'*holder* detiene il *lease*. La notifica avviene in maniera analoga a come illustrato in precedenza per l'uso di `O_ASYNC`: di default viene inviato al *lease holder* il segnale `SIGIO`, ma questo segnale può essere modificato usando il comando `F_SETSIG` di `fcntl` (anche in questo caso si può rispecificare lo stesso `SIGIO`).

Se si è fatto questo (ed in genere è opportuno farlo, come in precedenza, per utilizzare segnali *real-time*) e se inoltre si è installato il gestore del segnale con `SA_SIGINFO` si riceverà nel campo `si_fd` della struttura `siginfo_t` il valore del file descriptor del file sul quale è stato compiuto l'accesso; in questo modo un processo può mantenere anche più di un *file lease*.

Esistono due tipi di *file lease*: di lettura (*read lease*) e di scrittura (*write lease*). Nel primo caso la notifica avviene quando un altro processo esegue l'apertura del file in scrittura o usa `truncate` per troncarlo. Nel secondo caso la notifica avviene anche se il file viene aperto in

lettura; in quest'ultimo caso però il *lease* può essere ottenuto solo se nessun altro processo ha aperto lo stesso file.

Come accennato in sez. 5.2.5 il comando di `fcntl` che consente di acquisire un *file lease* è `F_SETLEASE`, che viene utilizzato anche per rilasciarlo. In tal caso il file descriptor `fd` passato a `fcntl` servirà come riferimento per il file su cui si vuole operare, mentre per indicare il tipo di operazione (acquisizione o rilascio) occorrerà specificare come valore dell'argomento `arg` di `fcntl` uno dei tre valori di tab. 10.9.

Valore	Significato
<code>F_RDLCK</code>	Richiede un <i>read lease</i> .
<code>F_WRLCK</code>	Richiede un <i>write lease</i> .
<code>F_UNLCK</code>	Rilascia un <i>file lease</i> .

Tabella 10.9: Costanti per i tre possibili valori dell'argomento `arg` di `fcntl` quando usata con i comandi `F_SETLEASE` e `F_GETLEASE`.

Se invece si vuole conoscere lo stato di eventuali *file lease* occorrerà chiamare `fcntl` sul relativo file descriptor `fd` con il comando `F_GETLEASE`, e si otterrà indietro nell'argomento `arg` uno dei valori di tab. 10.9, che indicheranno la presenza del rispettivo tipo di *lease*, o, nel caso di `F_UNLCK`, l'assenza di qualunque *file lease*.

Si tenga presente che un processo può mantenere solo un tipo di *lease* su un file, e che un *lease* può essere ottenuto solo su file di dati (*pipe* e dispositivi sono quindi esclusi). Inoltre un processo non privilegiato può ottenere un *lease* soltanto per un file appartenente ad un *UID* corrispondente a quello del processo. Soltanto un processo con privilegi di amministratore (cioè con la capacità `CAP_LEASE`, vedi sez. 9.1.1) può acquisire *lease* su qualunque file.

Se su un file è presente un *lease* quando il *lease breaker* esegue una `truncate` o una `open` che configge con esso,³⁵ la funzione si blocca (a meno di non avere aperto il file con `O_NONBLOCK`, nel qual caso `open` fallirebbe con un errore di `EWOULDBLOCK`) e viene eseguita la notifica al *lease holder*, così che questo possa completare le sue operazioni sul file e rilasciare il *lease*. In sostanza con un *read lease* si rilevano i tentativi di accedere al file per modificarne i dati da parte di un altro processo, mentre con un *write lease* si rilevano anche i tentativi di accesso in lettura. Si noti comunque che le operazioni di notifica avvengono solo in fase di apertura del file e non sulle singole operazioni di lettura e scrittura.

L'utilizzo dei *file lease* consente al *lease holder* di assicurare la consistenza di un file, a seconda dei due casi, prima che un altro processo inizi con le sue operazioni di scrittura o di lettura su di esso. In genere un *lease holder* che riceve una notifica deve provvedere a completare le necessarie operazioni (ad esempio scaricare eventuali buffer), per poi rilasciare il *lease* così che il *lease breaker* possa eseguire le sue operazioni. Questo si fa con il comando `F_SETLEASE`, o rimuovendo il *lease* con `F_UNLCK`, o, nel caso di *write lease* che configge con una operazione di lettura, declassando il *lease* a lettura con `F_RDLCK`.

Se il *lease holder* non provvede a rilasciare il *lease* entro il numero di secondi specificato dal parametro di sistema mantenuto in `/proc/sys/fs/lease-break-time` sarà il kernel stesso a rimuoverlo o declassarlo automaticamente (questa è una misura di sicurezza per evitare che un processo blocchi indefinitamente l'accesso ad un file acquisendo un *lease*). Una volta che un *lease* è stato rilasciato o declassato (che questo sia fatto dal *lease holder* o dal kernel è lo stesso) le chiamate a `open` o `truncate` eseguite dal *lease breaker* rimaste bloccate proseguono automaticamente.

³⁵in realtà `truncate` configge sempre, mentre `open`, se eseguita in sola lettura, non configge se si tratta di un *read lease*.

Benché possa risultare utile per sincronizzare l'accesso ad uno stesso file da parte di più processi, l'uso dei *file lease* non consente comunque di risolvere il problema di rilevare automaticamente quando un file o una directory vengono modificati,³⁶ che è quanto necessario ad esempio ai programmi di gestione dei file dei vari desktop grafici.

Per risolvere questo problema a partire dal kernel 2.4 è stata allora creata un'altra interfaccia,³⁷ chiamata *dnotify*, che consente di richiedere una notifica quando una directory, o uno qualunque dei file in essa contenuti, viene modificato. Come per i *file lease* la notifica avviene di default attraverso il segnale SIGIO, ma se ne può utilizzare un altro, e di nuovo, per le ragioni già esposte in precedenza, è opportuno che si utilizzino dei segnali *real-time*. Inoltre, come in precedenza, si potrà ottenere nel gestore del segnale il file descriptor che è stato modificato tramite il contenuto della struttura `siginfo_t`.

Valore	Significato
DN_ACCESS	Un file è stato acceduto, con l'esecuzione di una fra <code>read</code> , <code>pread</code> , <code>readv</code> .
DN_MODIFY	Un file è stato modificato, con l'esecuzione di una fra <code>write</code> , <code>pwrite</code> , <code>writv</code> , <code>truncate</code> , <code>ftruncate</code> .
DN_CREATE	È stato creato un file nella directory, con l'esecuzione di una fra <code>open</code> , <code>creat</code> , <code>mknod</code> , <code>mknod</code> , <code>link</code> , <code>symlink</code> , <code>rename</code> (da un'altra directory).
DN_DELETE	È stato cancellato un file dalla directory con l'esecuzione di una fra <code>unlink</code> , <code>rename</code> (su un'altra directory), <code>rmdir</code> .
DN_RENAME	È stato rinominato un file all'interno della directory (con <code>rename</code>).
DN_ATTRIB	È stato modificato un attributo di un file con l'esecuzione di una fra <code>chown</code> , <code>chmod</code> , <code>utime</code> .
DN_MULTISHOT	Richiede una notifica permanente di tutti gli eventi.

Tabella 10.10: Le costanti che identificano le varie classi di eventi per i quali si richiede la notifica con il comando `F_NOTIFY` di `fcntl`.

Ci si può registrare per le notifiche dei cambiamenti al contenuto di una certa directory eseguendo la funzione `fcntl` su un file descriptor associato alla stessa con il comando `F_NOTIFY`. In questo caso l'argomento `arg` di `fcntl` serve ad indicare per quali classi eventi si vuole ricevere la notifica, e prende come valore una maschera binaria composta dall'OR aritmetico di una o più delle costanti riportate in tab. 10.10.

A meno di non impostare in maniera esplicita una notifica permanente usando il valore `DN_MULTISHOT`, la notifica è singola: viene cioè inviata una sola volta quando si verifica un qualunque fra gli eventi per i quali la si è richiesta. Questo significa che un programma deve registrarsi un'altra volta se desidera essere notificato di ulteriori cambiamenti. Se si eseguono diverse chiamate con `F_NOTIFY` e con valori diversi per `arg` questi ultimi si *accumulano*; cioè eventuali nuovi classi di eventi specificate in chiamate successive vengono aggiunte a quelle già impostate nelle precedenti. Se si vuole rimuovere la notifica si deve invece specificare un valore nullo.

Il maggiore problema di *dnotify* è quello della scalabilità: si deve usare un file descriptor per ciascuna directory che si vuole tenere sotto controllo, il che porta facilmente ad avere un eccesso di file aperti. Inoltre quando la directory che si controlla è all'interno di un dispositivo

³⁶questa funzionalità venne aggiunta principalmente ad uso di Samba per poter facilitare l'emulazione del comportamento di Windows sui file, ma ad oggi viene considerata una interfaccia mal progettata ed il suo uso è fortemente sconsigliato a favore di *inotify*.

³⁷si ricordi che anche questa è una interfaccia specifica di Linux che deve essere evitata se si vogliono scrivere programmi portabili, e che le funzionalità illustrate sono disponibili soltanto se è stata definita la macro `_GNU_SOURCE`.

rimovibile, mantenere il relativo file descriptor aperto comporta l'impossibilità di smontare il dispositivo e di rimuoverlo, il che in genere complica notevolmente la gestione dell'uso di questi dispositivi.

Un altro problema è che l'interfaccia di *dnotify* consente solo di tenere sotto controllo il contenuto di una directory; la modifica di un file viene segnalata, ma poi è necessario verificare di quale file si tratta (operazione che può essere molto onerosa quando una directory contiene un gran numero di file). Infine l'uso dei segnali come interfaccia di notifica comporta tutti i problemi di gestione visti in sez. 7.3 e sez. 7.4. Per tutta questa serie di motivi in generale quella di *dnotify* viene considerata una interfaccia di usabilità problematica ed il suo uso oggi è fortemente sconsigliato.

Per risolvere i problemi appena illustrati è stata introdotta una nuova interfaccia per l'osservazione delle modifiche a file o directory, chiamata *inotify*.³⁸ Anche questa è una interfaccia specifica di Linux (pertanto non deve essere usata se si devono scrivere programmi portabili), ed è basata sull'uso di una coda di notifica degli eventi associata ad un singolo file descriptor, il che permette di risolvere il principale problema di *dnotify*. La coda viene creata attraverso la funzione di sistema *inotify_init*, il cui prototipo è:

```
#include <sys/inotify.h>
int inotify_init(void)
```

Inizializza una istanza di *inotify*.

La funzione ritorna un file descriptor in caso di successo, o `-1` in caso di errore, nel qual caso `errno` assumerà uno dei valori:

`EMFILE` si è raggiunto il numero massimo di istanze di *inotify* consentite all'utente.

`ENFILE` si è raggiunto il massimo di file descriptor aperti nel sistema.

`ENOMEM` non c'è sufficiente memoria nel kernel per creare l'istanza.

La funzione non prende alcun argomento; inizializza una istanza di *inotify* e restituisce un file descriptor attraverso il quale verranno effettuate le operazioni di notifica; si tratta di un file descriptor speciale che non è associato a nessun file su disco, e che viene utilizzato solo per notificare gli eventi che sono stati posti in osservazione. Per evitare abusi delle risorse di sistema è previsto che un utente possa utilizzare un numero limitato di istanze di *inotify*; il valore di default del limite è di 128, ma questo valore può essere cambiato con `sysctl` o usando il file `/proc/sys/fs/inotify/max_user_instances`.

Dato che questo file descriptor non è associato a nessun file o directory reale, l'inconveniente di non poter smontare un filesystem i cui file sono tenuti sotto osservazione viene completamente eliminato; anzi, una delle capacità dell'interfaccia di *inotify* è proprio quella di notificare il fatto che il filesystem su cui si trova il file o la directory osservata è stato smontato.

Inoltre trattandosi di un file descriptor a tutti gli effetti, esso potrà essere utilizzato come argomento per le funzioni `select` e `poll` e con l'interfaccia di *epoll*, ed a partire dal kernel 2.6.25 è stato introdotto anche il supporto per il `signal-driven I/O`. Siccome gli eventi vengono notificati come dati disponibili in lettura, dette funzioni ritorneranno tutte le volte che si avrà un evento di notifica.

Così, invece di dover utilizzare i segnali, considerati una pessima scelta dal punto di vista dell'interfaccia utente, si potrà gestire l'osservazione degli eventi con una qualunque delle modalità di *I/O multiplexing* illustrate in sez. 10.2. Qualora si voglia cessare l'osservazione, sarà sufficiente chiudere il file descriptor e tutte le risorse allocate saranno automaticamente

³⁸l'interfaccia è disponibile a partire dal kernel 2.6.13, le relative funzioni sono state introdotte nelle glibc 2.4.

rilasciate. Infine l'interfaccia di *inotify* consente di mettere sotto osservazione, oltre che una directory, anche singoli file.

Una volta creata la coda di notifica si devono definire gli eventi da tenere sotto osservazione; questo viene fatto attraverso una *lista di osservazione* (o *watch list*) che è associata alla coda. Per gestire la lista di osservazione l'interfaccia fornisce due funzioni di sistema, la prima di queste è *inotify_add_watch*, il cui prototipo è:

```
#include <sys/inotify.h>
int inotify_add_watch(int fd, const char *pathname, uint32_t mask)
    Aggiunge un evento di osservazione a una lista di osservazione.
```

La funzione ritorna un valore positivo in caso di successo, o `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EACCES` non si ha accesso in lettura al file indicato.

`EINVAL` `mask` non contiene eventi legali o `fd` non è un file descriptor di *inotify*.

`ENOSPC` si è raggiunto il numero massimo di voci di osservazione o il kernel non ha potuto allocare una risorsa necessaria.

ed inoltre `EFAULT`, `ENOMEM` e `EBADF` nel loro significato generico.

La funzione consente di creare un “osservatore” (il cosiddetto “*watch*”) nella lista di osservazione di una coda di notifica, che deve essere indicata specificando il file descriptor ad essa associato nell'argomento `fd`, che ovviamente dovrà essere un file descriptor creato con *inotify_init*. Il file o la directory da porre sotto osservazione vengono invece indicati per nome, da passare nell'argomento `pathname`. Infine il terzo argomento, `mask`, indica che tipo di eventi devono essere tenuti sotto osservazione e le modalità della stessa. L'operazione può essere ripetuta per tutti i file e le directory che si vogliono tenere sotto osservazione,³⁹ e si utilizzerà sempre un solo file descriptor.

Il tipo di evento che si vuole osservare deve essere specificato nell'argomento `mask` come maschera binaria, combinando i valori delle costanti riportate in tab. 10.11 che identificano i singoli bit della maschera ed il relativo significato. In essa si sono marcati con un “●” gli eventi che, quando specificati per una directory, vengono osservati anche su tutti i file che essa contiene. Nella seconda parte della tabella si sono poi indicate alcune combinazioni predefinite dei flag della prima parte.

Oltre ai flag di tab. 10.11, che indicano il tipo di evento da osservare e che vengono utilizzati anche in uscita per indicare il tipo di evento avvenuto, *inotify_add_watch* supporta ulteriori flag,⁴⁰ riportati in tab. 10.12, che indicano le modalità di osservazione (da passare sempre nell'argomento `mask`) e che al contrario dei precedenti non vengono mai impostati nei risultati in uscita.

Se non esiste nessun *watch* per il file o la directory specificata questo verrà creato per gli eventi specificati dall'argomento `mask`, altrimenti la funzione sovrascriverà le impostazioni precedenti, a meno che non si sia usato il flag `IN_MASK_ADD`, nel qual caso gli eventi specificati saranno aggiunti a quelli già presenti.

Come accennato quando si tiene sotto osservazione una directory vengono restituite le informazioni sia riguardo alla directory stessa che ai file che essa contiene; questo comportamento può essere disabilitato utilizzando il flag `IN_ONLYDIR`, che richiede di riportare soltanto gli eventi relativi alla directory stessa. Si tenga presente inoltre che quando si osserva una directory vengono riportati solo gli eventi sui file che essa contiene direttamente, non quelli

³⁹anche in questo caso c'è un limite massimo che di default è pari a 8192, ed anche questo valore può essere cambiato con `sysctl` o usando il file `/proc/sys/fs/inotify/max_user_watches`.

⁴⁰i flag `IN_DONT_FOLLOW`, `IN_MASK_ADD` e `IN_ONLYDIR` sono stati introdotti a partire dalle glibc 2.5, se si usa la versione 2.4 è necessario definirli a mano.

Valore	Significato
IN_ACCESS	• C'è stato accesso al file in lettura.
IN_ATTRIB	• Ci sono stati cambiamenti sui dati dell' <i>inode</i> (o sugli attributi estesi, vedi sez. 4.5.1).
IN_CLOSE_WRITE	• È stato chiuso un file aperto in scrittura.
IN_CLOSE_NOWRITE	• È stato chiuso un file aperto in sola lettura.
IN_CREATE	• È stato creato un file o una directory in una directory sotto osservazione.
IN_DELETE	• È stato cancellato un file o una directory in una directory sotto osservazione.
IN_DELETE_SELF	– È stato cancellato il file (o la directory) sotto osservazione.
IN_MODIFY	• È stato modificato il file.
IN_MOVE_SELF	È stato rinominato il file (o la directory) sotto osservazione.
IN_MOVED_FROM	• Un file è stato spostato fuori dalla directory sotto osservazione.
IN_MOVED_TO	• Un file è stato spostato nella directory sotto osservazione.
IN_OPEN	• Un file è stato aperto.
IN_CLOSE	Combinazione di IN_CLOSE_WRITE e IN_CLOSE_NOWRITE.
IN_MOVE	Combinazione di IN_MOVED_FROM e IN_MOVED_TO.
IN_ALL_EVENTS	Combinazione di tutti i flag possibili.

Tabella 10.11: Le costanti che identificano i bit della maschera binaria dell'argomento `mask` di `inotify_add_watch` che indicano il tipo di evento da tenere sotto osservazione.

Valore	Significato
IN_DONT_FOLLOW	Non dereferenzia <code>pathname</code> se questo è un link simbolico.
IN_MASK_ADD	Aggiunge a quelli già impostati i flag indicati nell'argomento <code>mask</code> , invece di sovrascriverli.
IN_ONESHOT	Esegue l'osservazione su <code>pathname</code> per una sola volta, rimuovendolo poi dalla <i>watch list</i> .
IN_ONLYDIR	Se <code>pathname</code> è una directory riporta soltanto gli eventi ad essa relativi e non quelli per i file che contiene.

Tabella 10.12: Le costanti che identificano i bit della maschera binaria dell'argomento `mask` di `inotify_add_watch` che indicano le modalità di osservazione.

relativi a file contenuti in eventuali sottodirectory; se si vogliono osservare anche questi sarà necessario creare ulteriori *watch* per ciascuna sottodirectory.

Infine usando il flag `IN_ONESHOT` è possibile richiedere una notifica singola;⁴¹ una volta verificatosi uno qualunque fra gli eventi richiesti con `inotify_add_watch` l'osservatore verrà automaticamente rimosso dalla lista di osservazione e nessun ulteriore evento sarà più notificato.

In caso di successo `inotify_add_watch` ritorna un intero positivo, detto *watch descriptor*, che identifica univocamente un osservatore su una coda di notifica; esso viene usato per farvi riferimento sia riguardo i risultati restituiti da *inotify*, che per la eventuale rimozione dello stesso.

La seconda funzione di sistema per la gestione delle code di notifica, che permette di rimuovere un osservatore, è `inotify_rm_watch`, ed il suo prototipo è:

```
#include <sys/inotify.h>
int inotify_rm_watch(int fd, uint32_t wd)
```

Rimuove un osservatore da una coda di notifica.

⁴¹questa funzionalità però è disponibile soltanto a partire dal kernel 2.6.16.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` non si è specificato in `fd` un file descriptor valido.

`EINVAL` il valore di `wd` non è corretto, o `fd` non è associato ad una coda di notifica.

La funzione rimuove dalla coda di notifica identificata dall'argomento `fd` l'osservatore identificato dal *watch descriptor* `wd`; ovviamente deve essere usato per questo argomento un valore ritornato da `inotify_add_watch`, altrimenti si avrà un errore di `EINVAL`. In caso di successo della rimozione, contemporaneamente alla cancellazione dell'osservatore, sulla coda di notifica verrà generato un evento di tipo `IN_IGNORED` (vedi tab. 10.13). Si tenga presente che se un file viene cancellato o un filesystem viene smontato i relativi osservatori vengono rimossi automaticamente e non è necessario utilizzare `inotify_rm_watch`.

Come accennato l'interfaccia di *inotify* prevede che gli eventi siano notificati come dati presenti in lettura sul file descriptor associato alla coda di notifica. Una applicazione pertanto dovrà leggere i dati da detto file con una `read`, che ritornerà sul buffer i dati presenti nella forma di una o più strutture di tipo `inotify_event` (la cui definizione è riportata in fig. 10.12). Qualora non siano presenti dati la `read` si bloccherà (a meno di non aver impostato il file descriptor in modalità non bloccante) fino all'arrivo di almeno un evento.

```

struct inotify_event {
    int      wd;          /* Watch descriptor */
    uint32_t mask;        /* Mask of events */
    uint32_t cookie;      /* Unique cookie associating related
                           events (for rename(2)) */
    uint32_t len;         /* Size of 'name' field */
    char     name[];      /* Optional null-terminated name */
};

```

Figura 10.12: La struttura `inotify_event` usata dall'interfaccia di *inotify* per riportare gli eventi.

Una ulteriore caratteristica dell'interfaccia di *inotify* è che essa permette di ottenere con `ioctl`, come per i file descriptor associati ai socket (si veda sez. 16.3.3), il numero di byte disponibili in lettura sul file descriptor, utilizzando su di esso l'operazione `FIONREAD`.⁴² Si può così utilizzare questa operazione, oltre che per predisporre una operazione di lettura con un buffer di dimensioni adeguate, anche per ottenere rapidamente il numero di file che sono cambiati.

Una volta effettuata la lettura con `read` a ciascun evento sarà associata una struttura `inotify_event` contenente i rispettivi dati. Per identificare a quale file o directory l'evento corrisponde viene restituito nel campo `wd` il *watch descriptor* con cui il relativo osservatore è stato registrato. Il campo `mask` contiene invece una maschera di bit che identifica il tipo di evento verificatosi; in essa compariranno sia i bit elencati nella prima parte di tab. 10.11, che gli eventuali valori aggiuntivi di tab. 10.13 (questi compaiono solo nel campo `mask` di `inotify_event`, e non sono utilizzabili in fase di registrazione dell'osservatore).

⁴²questa è una delle operazioni speciali per i file (vedi sez. 5.2.5), che è disponibile solo per i socket e per i file descriptor creati con `inotify_init`.

⁴³la coda di notifica ha una dimensione massima che viene controllata dal parametro di sistema `/proc/sys/fs/inotify/max_queued_events`, che indica il numero massimo di eventi che possono essere mantenuti sulla stessa; quando detto valore viene ecceduto gli ulteriori eventi vengono scartati, ma viene comunque generato un evento di tipo `IN_Q_OVERFLOW`.

Valore	Significato
IN_IGNORED	L'osservatore è stato rimosso, sia in maniera esplicita con l'uso di <code>inotify_rm_watch</code> , che in maniera implicita per la rimozione dell'oggetto osservato o per lo smontaggio del filesystem su cui questo si trova.
IN_ISDIR	L'evento avvenuto fa riferimento ad una directory (consente così di distinguere, quando si pone sotto osservazione una directory, fra gli eventi relativi ad essa e quelli relativi ai file che essa contiene).
IN_Q_OVERFLOW	Si sono eccedute le dimensioni della coda degli eventi (<i>overflow</i> della coda); in questo caso il valore di <code>wd</code> è <code>-1</code> . ⁴³
IN_UNMOUNT	Il filesystem contenente l'oggetto posto sotto osservazione è stato smontato.

Tabella 10.13: Le costanti che identificano i bit aggiuntivi usati nella maschera binaria del campo `mask` di `inotify_event`.

Il campo `cookie` contiene invece un intero univoco che permette di identificare eventi correlati (per i quali avrà lo stesso valore), al momento viene utilizzato soltanto per rilevare lo spostamento di un file, consentendo così all'applicazione di collegare la corrispondente coppia di eventi `IN_MOVED_TO` e `IN_MOVED_FROM`.

Infine due campi `name` e `len` sono utilizzati soltanto quando l'evento è relativo ad un file presente in una directory posta sotto osservazione, in tal caso essi contengono rispettivamente il nome del file (come *pathname* relativo alla directory osservata) e la relativa dimensione in byte. Il campo `name` viene sempre restituito come stringa terminata da `NUL`, con uno o più zeri di terminazione, a seconda di eventuali necessità di allineamento del risultato, ed il valore di `len` corrisponde al totale della dimensione di `name`, zeri aggiuntivi compresi. La stringa con il nome del file viene restituita nella lettura subito dopo la struttura `inotify_event`; questo significa che le dimensioni di ciascun evento di *inotify* saranno pari a `sizeof(inotify_event) + len`.

Vediamo allora un esempio dell'uso dell'interfaccia di *inotify* con un semplice programma che permette di mettere sotto osservazione uno o più file e directory. Il programma si chiama `inotify_monitor.c` ed il codice completo è disponibile coi sorgenti allegati alla guida, il corpo principale del programma, che non contiene la sezione di gestione delle opzioni e le funzioni di ausilio è riportato in fig. 10.13.

Una volta completata la scansione delle opzioni il corpo del programma inizia controllando (11-15) che sia rimasto almeno un argomento che indichi quale file o directory mettere sotto osservazione (e qualora questo non avvenga esce stampando la pagina di aiuto); dopo di che passa (16-20) all'inizializzazione di *inotify* ottenendo con `inotify_init` il relativo file descriptor (o si esce in caso di errore).

Il passo successivo è aggiungere (21-30) alla coda di notifica gli opportuni osservatori per ciascuno dei file o directory indicati all'invocazione del comando; questo viene fatto eseguendo un ciclo (22-29) fintanto che la variabile `i`, inizializzata a zero (21) all'inizio del ciclo, è minore del numero totale di argomenti rimasti. All'interno del ciclo si invoca (23) `inotify_add_watch` per ciascuno degli argomenti, usando la maschera degli eventi data dalla variabile `mask` (il cui valore viene impostato nella scansione delle opzioni), in caso di errore si esce dal programma altrimenti si incrementa l'indice (29).

Completa l'inizializzazione di *inotify* inizia il ciclo principale (32-56) del programma, nel quale si resta in attesa degli eventi che si intendono osservare. Questo viene fatto eseguendo all'inizio del ciclo (33) una `read` che si bloccherà fintanto che non si saranno verificati eventi.

Dato che l'interfaccia di *inotify* può riportare anche più eventi in una sola lettura, si è avuto cura di passare alla `read` un buffer di dimensioni adeguate, inizializzato in (7) ad un valore di approssimativamente 512 eventi (si ricordi che la quantità di dati restituita da *inotify* è variabile a causa della diversa lunghezza del nome del file restituito insieme a `inotify_event`).

```

1 #include <sys/inotify.h> /* Linux inotify interface */
2 ...
3 int main(int argc, char *argv[])
4 {
5     int i, narg, nread;
6     int fd, wd;
7     char buffer[512 * (sizeof(struct inotify_event) + 16)];
8     unsigned int mask=0;
9     struct inotify_event * event;
10    ...
11    narg = argc - optind;
12    if (narg < 1) { /* There must be at least one argument */
13        printf("Wrong_number_of_arguments_%d\n", argc - optind);
14        usage();
15    }
16    fd = inotify_init(); /* initialize inotify */
17    if (fd < 0) {
18        perror("Failing_on_inotify_init");
19        exit(-1);
20    }
21    i = 0;
22    while (i < narg) {
23        wd = inotify_add_watch(fd, argv[optind+i], mask); /* add watch */
24        if (wd <= 0) {
25            printf("Failing_to_add_watched_file_%s,_mask_%i;_%s\n",
26                argv[optind+i], mask, strerror(errno));
27            exit(-1);
28        }
29        i++;
30    }
31    /* Main Loop: read events and print them */
32    while (1) {
33        nread = read(fd, buffer, sizeof(buffer));
34        if (nread < 0) {
35            if (errno == EINTR) {
36                continue;
37            } else {
38                perror("error_reading_inotify_data");
39                exit(1);
40            }
41        } else {
42            i = 0;
43            while (i < nread) {
44                event = (struct inotify_event *) buffer + i;
45                printf("Watch_descriptor_%i\n", event->wd);
46                printf("Observed_event_on_%s\n", argv[optind-1+event->wd]);
47                if (event->len) {
48                    printf("On_file_%s\n", event->name);
49                }
50                printevent(event->mask);
51                i += sizeof(struct inotify_event) + event->len;
52            }
53        }
54    }
55    return 0;
56 }

```

Figura 10.13: Esempio di codice che usa l'interfaccia di *inotify*.

In caso di errore di lettura (35-40) il programma esce con un messaggio di errore (37-39), a meno che non si tratti di una interruzione della *system call*, nel qual caso (36) si ripete la lettura.

Se la lettura è andata a buon fine invece si esegue un ciclo (43-52) per leggere tutti gli eventi restituiti, al solito si inizializza l'indice *i* a zero (42) e si ripetono le operazioni (43) fintanto che esso non supera il numero di byte restituiti in lettura. Per ciascun evento all'interno del ciclo si assegna alla variabile *event* (si noti come si sia eseguito un opportuno *casting* del puntatore) l'indirizzo nel buffer della corrispondente struttura *inotify_event* (44), e poi si stampano il numero di *watch descriptor* (45) ed il file a cui questo fa riferimento (46), ricavato dagli argomenti passati a riga di comando sfruttando il fatto che i *watch descriptor* vengono assegnati in ordine progressivo crescente a partire da 1.

Qualora sia presente il riferimento ad un nome di file associato all'evento lo si stampa (47-49); si noti come in questo caso si sia controllato il valore del campo *event->len* e non il fatto che *event->name* riporti o meno un puntatore nullo. L'interfaccia infatti, qualora il nome non sia presente, non tocca il campo *event->name*, che si troverà pertanto a contenere quello che era precedentemente presente nella rispettiva locazione di memoria, nel caso più comune il puntatore al nome di un file osservato in precedenza.

Si utilizza poi (50) la funzione *prntevent*, che interpreta il valore del campo *event->mask*, per stampare il tipo di eventi accaduti.⁴⁴ Infine (51) si provvede ad aggiornare l'indice *i* per farlo puntare all'evento successivo.

Se adesso usiamo il programma per mettere sotto osservazione una directory, e da un altro terminale eseguiamo il comando *ls* otterremo qualcosa del tipo di:

```
piccardi@gethen:~/gapil/sources$ ./inotify_monitor -a /home/piccardi/gapil/
Watch descriptor 1
Observed event on /home/piccardi/gapil/
IN_OPEN,
Watch descriptor 1
Observed event on /home/piccardi/gapil/
IN_CLOSE_NOWRITE,
```

I lettori più accorti si saranno resi conto che nel ciclo di lettura degli eventi appena illustrato non viene trattato il caso particolare in cui la funzione *read* restituisce in *nread* un valore nullo. Lo si è fatto perché con *inotify* il ritorno di una *read* con un valore nullo avviene soltanto, come forma di avviso, quando si sia eseguita la funzione specificando un buffer di dimensione insufficiente a contenere anche un solo evento. Nel nostro caso le dimensioni erano senz'altro sufficienti, per cui tale evenienza non si verificherà mai.

Ci si potrà però chiedere cosa succede se il buffer è sufficiente per un evento, ma non per tutti gli eventi verificatisi. Come si potrà notare nel codice illustrato in precedenza non si è presa nessuna precauzione per verificare che non ci fossero stati troncamenti dei dati. Anche in questo caso il comportamento scelto è corretto, perché l'interfaccia di *inotify* garantisce automaticamente, anche quando ne sono presenti in numero maggiore, di restituire soltanto il numero di eventi che possono rientrare completamente nelle dimensioni del buffer specificato.⁴⁵ Se gli eventi sono di più saranno restituiti solo quelli che entrano interamente nel buffer e gli altri saranno restituiti alla successiva chiamata di *read*.

Infine un'ultima caratteristica dell'interfaccia di *inotify* è che gli eventi restituiti nella lettura formano una sequenza ordinata, è cioè garantito che se si esegue uno spostamento di

⁴⁴per il relativo codice, che non riportiamo in quanto non essenziale alla comprensione dell'esempio, si possono utilizzare direttamente i sorgenti allegati alla guida.

⁴⁵si avrà cioè, facendo riferimento sempre al codice di fig. 10.13, che *read* sarà in genere minore delle dimensioni di buffer ed uguale soltanto qualora gli eventi corrispondano esattamente alle dimensioni di quest'ultimo.

un file gli eventi vengano generati nella sequenza corretta. L'interfaccia garantisce anche che se si verificano più eventi consecutivi identici (vale a dire con gli stessi valori dei campi `wd`, `mask`, `cookie`, e `name`) questi vengono raggruppati in un solo evento.

10.3.3 L'interfaccia POSIX per l'I/O asincrono

Una modalità alternativa all'uso dell'I/O *multiplexing* per gestione dell'I/O simultaneo su molti file è costituita dal cosiddetto I/O *asincrono* o "AIO". Il concetto base dell'I/O *asincrono* è che le funzioni di I/O non attendono il completamento delle operazioni prima di ritornare, così che il processo non viene bloccato. In questo modo diventa ad esempio possibile effettuare una richiesta preventiva di dati, in modo da poter effettuare in contemporanea le operazioni di calcolo e quelle di I/O.

Benché la modalità di apertura asincrona di un file vista in sez. 10.3.1 possa risultare utile in varie occasioni (in particolar modo con i socket e gli altri file per i quali le funzioni di I/O sono *system call* lente), essa è comunque limitata alla notifica della disponibilità del file descriptor per le operazioni di I/O, e non ad uno svolgimento asincrono delle medesime. Lo standard POSIX.1b definisce una interfaccia apposita per l'I/O asincrono vero e proprio,⁴⁶ che prevede un insieme di funzioni dedicate per la lettura e la scrittura dei file, completamente separate rispetto a quelle usate normalmente.

In generale questa interfaccia è completamente astratta e può essere implementata sia direttamente nel kernel che in *user space* attraverso l'uso di *thread*. Per le versioni del kernel meno recenti esiste una implementazione di questa interfaccia fornita completamente dalla *glibc* a partire dalla versione 2.1, che è realizzata completamente in *user space*, ed è accessibile linkando i programmi con la libreria `librt`. A partire dalla versione 2.5.32 è stato introdotto nel kernel una nuova infrastruttura per l'I/O asincrono, ma ancora il supporto è parziale ed insufficiente ad implementare tutto l'AIO POSIX.

Lo standard POSIX prevede che tutte le operazioni di I/O asincrono siano controllate attraverso l'uso di una apposita struttura `aiocb` (il cui nome sta per *asynchronous I/O control block*), che viene passata come argomento a tutte le funzioni dell'interfaccia. La sua definizione, come effettuata in `aio.h`, è riportata in fig. 10.14. Nello stesso file è definita la macro `_POSIX_ASYNCHRONOUS_IO`, che dichiara la disponibilità dell'interfaccia per l'I/O asincrono.

```

struct aiocb
{
    int aio_fildes;           /* File descriptor. */
    off_t aio_offset;        /* File offset */
    volatile void *aio_buf;  /* Location of buffer. */
    size_t aio_nbytes;       /* Length of transfer. */
    int aio_reqprio;         /* Request priority offset. */
    struct sigevent aio_sigevent; /* Signal number and value. */
    int aio_lio_opcode;      /* Operation to be performed. */
};

```

Figura 10.14: La struttura `aiocb`, usata per il controllo dell'I/O asincrono.

Le operazioni di I/O asincrono possono essere effettuate solo su un file già aperto; il file deve inoltre supportare la funzione `lseek`, pertanto terminali e *pipe* sono esclusi. Non c'è limite al numero di operazioni contemporanee effettuabili su un singolo file. Ogni operazione

⁴⁶questa è stata ulteriormente perfezionata nelle successive versioni POSIX.1-2001 e POSIX.1-2008.

deve inizializzare opportunamente un *control block*. Il file descriptor su cui operare deve essere specificato tramite il campo `aio_fildes`; dato che più operazioni possono essere eseguita in maniera asincrona, il concetto di posizione corrente sul file viene a mancare; pertanto si deve sempre specificare nel campo `aio_offset` la posizione sul file da cui i dati saranno letti o scritti. Nel campo `aio_buf` deve essere specificato l'indirizzo del buffer usato per l'I/O, ed in `aio_nbytes` la lunghezza del blocco di dati da trasferire.

Il campo `aio_reqprio` permette di impostare la priorità delle operazioni di I/O, in generale perché ciò sia possibile occorre che la piattaforma supporti questa caratteristica, questo viene indicato dal fatto che le macro `_POSIX_PRIORITY_IO`, e `_POSIX_PRIORITY_SCHEDULING` sono definite. La priorità viene impostata a partire da quella del processo chiamante (vedi sez. 3.3), cui viene sottratto il valore di questo campo. Il campo `aio_lio_opcode` è usato solo dalla funzione `lio_listio`, che, come vedremo, permette di eseguire con una sola chiamata una serie di operazioni, usando un vettore di *control block*. Tramite questo campo si specifica quale è la natura di ciascuna di esse.

Infine il campo `aio_sigevent` è una struttura di tipo `sigevent` (illustrata in fig. 7.15) che serve a specificare il modo in cui si vuole che venga effettuata la notifica del completamento delle operazioni richieste; per la trattazione delle modalità di utilizzo della stessa si veda quanto già visto in proposito in sez. 7.5.2.

Le due funzioni base dell'interfaccia per l'I/O asincrono sono `aio_read` ed `aio_write`. Esse permettono di richiedere una lettura od una scrittura asincrona di dati usando la struttura `aiocb` appena descritta; i rispettivi prototipi sono:

<pre>#include <aio.h> int aio_read(struct aiocb *aiocbp) int aio_write(struct aiocb *aiocbp)</pre>	<p>Richiede una lettura asincrona.</p> <p>Richiede una scrittura asincrona.</p>
<p>Le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EAGAIN la coda delle richieste è momentaneamente piena.</p> <p>EBADF si è specificato un file descriptor sbagliato.</p> <p>EINVAL si è specificato un valore non valido per i campi <code>aio_offset</code> o <code>aio_reqprio</code> di <code>aiocbp</code>.</p> <p>ENOSYS la funzione non è implementata.</p>	

Entrambe le funzioni ritornano immediatamente dopo aver messo in coda la richiesta, o in caso di errore. Non è detto che gli errori `EBADF` ed `EINVAL` siano rilevati immediatamente al momento della chiamata, potrebbero anche emergere nelle fasi successive delle operazioni. Lettura e scrittura avvengono alla posizione indicata da `aio_offset`, a meno che il file non sia stato aperto in *append mode* (vedi sez. 5.1.2), nel qual caso le scritture vengono effettuate comunque alla fine del file, nell'ordine delle chiamate a `aio_write`.

Si tenga inoltre presente che deallocare la memoria indirizzata da `aiocbp` o modificarne i valori prima della conclusione di una operazione può dar luogo a risultati imprevedibili, perché l'accesso ai vari campi per eseguire l'operazione può avvenire in un momento qualsiasi dopo la richiesta. Questo comporta che non si devono usare per `aiocbp` variabili automatiche e che non si deve riutilizzare la stessa struttura per un'altra operazione fintanto che la precedente non sia stata ultimata. In generale per ogni operazione si deve utilizzare una diversa struttura `aiocb`.

Dato che si opera in modalità asincrona, il successo di `aio_read` o `aio_write` non implica che le operazioni siano state effettivamente eseguite in maniera corretta; per verificarne l'esito l'interfaccia prevede altre due funzioni, che permettono di controllare lo stato di esecuzione.

La prima è `aio_error`, che serve a determinare un eventuale stato di errore; il suo prototipo è:

```
#include <aio.h>
int aio_error(const struct aiocb *aiocbp)
    Determina lo stato di errore di una operazione di I/O asincrono.
```

La funzione ritorna 0 se le operazioni si sono concluse con successo, altrimenti restituisce EINPROGRESS se non sono concluse, ECANCELED se sono state cancellate o il relativo codice di errore se sono fallite.

Se l'operazione non si è ancora completata viene sempre restituito l'errore di EINPROGRESS, mentre se è stata cancellata ritorna ECANCELED. La funzione ritorna zero quando l'operazione si è conclusa con successo, altrimenti restituisce il codice dell'errore verificatosi, ed esegue la corrispondente impostazione di `errno`. Il codice può essere sia EINVAL ed EBADF, dovuti ad un valore errato per `aiocbp`, che uno degli errori possibili durante l'esecuzione dell'operazione di I/O richiesta, nel qual caso saranno restituiti, a seconda del caso, i codici di errore delle *system call* `read`, `write`, `fsync` e `fdatasync`.

Una volta che si sia certi che le operazioni siano state concluse (cioè dopo che una chiamata ad `aio_error` non ha restituito EINPROGRESS), si potrà usare la funzione `aio_return`, che permette di verificare il completamento delle operazioni di I/O asincrono; il suo prototipo è:

```
#include <aio.h>
ssize_t aio_return(const struct aiocb *aiocbp)
    Ottiene lo stato dei risultati di una operazione di I/O asincrono.
```

La funzione ritorna lo stato di uscita dell'operazione eseguita (il valore che avrebbero restituito le equivalenti funzioni eseguite in maniera sincrona).

La funzione recupera il valore dello stato di ritorno delle operazioni di I/O associate a `aiocbp` e deve essere chiamata una sola volta per ciascuna operazione asincrona, essa infatti fa sì che il sistema rilasci le risorse ad essa associate. É per questo motivo che occorre chiamare la funzione solo dopo che l'operazione cui `aiocbp` fa riferimento si è completata verificandolo con `aio_error`, ed usarla una sola volta. Una chiamata precedente il completamento delle operazioni darebbe risultati indeterminati, così come chiamarla più di una volta.

La funzione restituisce il valore di ritorno relativo all'operazione eseguita, così come ricavato dalla sottostante *system call* (il numero di byte letti, scritti o il valore di ritorno di `fsync` o `fdatasync`). É importante chiamare sempre questa funzione, altrimenti le risorse disponibili per le operazioni di I/O asincrono non verrebbero liberate, rischiando di arrivare ad un loro esaurimento.

Oltre alle operazioni di lettura e scrittura l'interfaccia POSIX.1b mette a disposizione un'altra operazione, quella di sincronizzazione dell'I/O, compiuta dalla funzione `aio_fsync`, che ha lo stesso effetto della analoga `fsync`, ma viene eseguita in maniera asincrona; il suo prototipo è:

```
#include <aio.h>
int aio_fsync(int op, struct aiocb *aiocbp)
    Richiede la sincronizzazione dei dati su disco.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà gli stessi valori visti `aio_read` con lo stesso significato.

La funzione richiede la sincronizzazione dei dati delle operazioni di I/O relative al file descriptor indicato in `aiocbp->aio_fildes`, ritornando immediatamente. Si tenga presente che la funzione mette semplicemente in coda la richiesta, l'esecuzione effettiva della sincronizzazione dovrà essere verificata con `aio_error` e `aio_return` come per le operazioni di lettura e scrittura. L'argomento `op` permette di indicare la modalità di esecuzione, se si specifica il

valore `O_DSYNC` le operazioni saranno completate con una chiamata a `fdatasync`, se si specifica `O_SYNC` con una chiamata a `fsync` (per i dettagli vedi sez. 5.2.3).

Il successo della chiamata assicura la richiesta di sincronizzazione dei dati relativi operazioni di I/O asincrono richieste fino a quel momento, niente è garantito riguardo la sincronizzazione dei dati relativi ad eventuali operazioni richieste successivamente. Se si è specificato un meccanismo di notifica questo sarà innescato una volta che le operazioni di sincronizzazione dei dati saranno completate (`aio_sigevent` è l'unico altro campo di `aiocbp` che viene usato).

In alcuni casi può essere necessario interrompere le operazioni di I/O (in genere quando viene richiesta un'uscita immediata dal programma), per questo lo standard POSIX.1b prevede una funzione apposita, `aio_cancel`, che permette di cancellare una operazione richiesta in precedenza; il suo prototipo è:

```
#include <aio.h>
int aio_cancel(int fd, struct aiocb *aiocbp)
    Richiede la cancellazione delle operazioni di I/O asincrono.
```

La funzione ritorna un intero positivo che indica il risultato dell'operazione in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` `fd` non è un file descriptor valido.

`ENOSYS` la funzione non è implementata.

La funzione permette di cancellare una operazione specifica sul file `fd`, indicata con `aiocbp`, o tutte le operazioni pendenti, specificando `NULL` come valore di `aiocbp`. Quando una operazione viene cancellata una successiva chiamata ad `aio_error` riporterà `ECANCELED` come codice di errore, ed mentre il valore di ritorno per `aio_return` sarà `-1`, inoltre il meccanismo di notifica non verrà invocato. Se con `aiocbp` si specifica una operazione relativa ad un file descriptor diverso da `fd` il risultato è indeterminato. In caso di successo, i possibili valori di ritorno per `aio_cancel` (anch'essi definiti in `aio.h`) sono tre:

`AIO_ALLDONE` indica che le operazioni di cui si è richiesta la cancellazione sono state già completate,

`AIO_CANCELED` indica che tutte le operazioni richieste sono state cancellate,

`AIO_NOTCANCELED` indica che alcune delle operazioni erano in corso e non sono state cancellate.

Nel caso si abbia `AIO_NOTCANCELED` occorrerà chiamare `aio_error` per determinare quali sono le operazioni effettivamente cancellate. Le operazioni che non sono state cancellate proseguiranno il loro corso normale, compreso quanto richiesto riguardo al meccanismo di notifica del loro avvenuto completamento.

Benché l'I/O asincrono preveda un meccanismo di notifica, l'interfaccia fornisce anche una apposita funzione, `aio_suspend`, che permette di sospendere l'esecuzione del processo chiamante fino al completamento di una specifica operazione; il suo prototipo è:

```
#include <aio.h>
int aio_suspend(const struct aiocb * const list[], int nent,
    const struct timespec *timeout)
    Attende il completamento di una operazione di I/O asincrono.
```

La funzione ritorna 0 se una (o più) operazioni sono state completate e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` nessuna operazione è stata completata entro `timeout`.

`EINTR` la funzione è stata interrotta da un segnale.

`ENOSYS` la funzione non è implementata.

La funzione permette di bloccare il processo fintanto che almeno una delle *nent* operazioni specificate nella lista *list* è completata, per un tempo massimo specificato dalla struttura *timespec* puntata da *timeout*, o fintanto che non arrivi un segnale (si tenga conto che questo segnale potrebbe essere anche quello utilizzato come meccanismo di notifica). La lista deve essere inizializzata con delle strutture *aiocb* relative ad operazioni effettivamente richieste, ma può contenere puntatori nulli, che saranno ignorati. In caso si siano specificati valori non validi l'effetto è indefinito. Un valore *NULL* per *timeout* comporta l'assenza di timeout, mentre se si vuole effettuare un *polling* sulle operazioni occorrerà specificare un puntatore valido ad una struttura *timespec* (vedi fig. 4.16) contenente valori nulli, e verificare poi con *aio_error* quale delle operazioni della lista *list* è stata completata.

Lo standard POSIX.1b infine ha previsto pure una funzione, *lio_listio*, che permette di effettuare la richiesta di una intera lista di operazioni di lettura o scrittura; il suo prototipo è:

```
#include <aio.h>
int lio_listio(int mode, struct aiocb * const list[], int nent, struct sigevent *sig)
```

Richiede l'esecuzione di una serie di operazioni di I/O.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso *errno* assumerà uno dei valori:

EAGAIN	nessuna operazione è stata completata entro timeout.
EINTR	la funzione è stata interrotta da un segnale.
EINVAL	si è passato un valore di <i>mode</i> non valido o un numero di operazioni <i>nent</i> maggiore di <i>AIO_LISTIO_MAX</i> .
ENOSYS	la funzione non è implementata.

La funzione esegue la richiesta delle *nent* operazioni indicate nella lista *list* un vettore di puntatori a strutture *aiocb* indicanti le operazioni da compiere (che verranno eseguite senza un ordine particolare). La lista può contenere anche puntatori nulli, che saranno ignorati (si possono così eliminare facilmente componenti della lista senza doverla rigenerare).

Ciascuna struttura *aiocb* della lista deve contenere un *control block* opportunamente inizializzato; in particolare per ognuna di esse dovrà essere specificato il tipo di operazione con il campo *aio_lio_opcode*, che può prendere i valori:

LIO_READ si richiede una operazione di lettura.

LIO_WRITE si richiede una operazione di scrittura.

LIO_NOP non si effettua nessuna operazione.

dove **LIO_NOP** viene usato quando si ha a che fare con un vettore di dimensione fissa, per poter specificare solo alcune operazioni, o quando si sono dovute cancellare delle operazioni e si deve ripetere la richiesta per quelle non completate.

L'argomento *mode* controlla il comportamento della funzione, se viene usato il valore **LIO_WAIT** la funzione si blocca fino al completamento di tutte le operazioni richieste; se si usa **LIO_NOWAIT** la funzione ritorna immediatamente dopo aver messo in coda tutte le richieste. In tal caso il chiamante può richiedere la notifica del completamento di tutte le richieste, impostando l'argomento *sig* in maniera analoga a come si fa per il campo *aio_sigevent* di *aiocb*.

10.4 Altre modalità di I/O avanzato

Oltre alle precedenti modalità di *I/O multiplexing* e *I/O asincrono*, esistono altre funzioni che implementano delle modalità di accesso ai file più evolute rispetto alle normali funzioni di lettura e scrittura che abbiamo esaminato in sez. 5.1. In questa sezione allora prenderemo in esame le interfacce per l'*I/O mappato in memoria*, per l'*I/O vettorizzato* e altre funzioni di I/O avanzato.

10.4.1 File mappati in memoria

Una modalità alternativa di I/O, che usa una interfaccia completamente diversa rispetto a quella classica vista in sez. 5.1, è il cosiddetto *memory-mapped I/O*, che attraverso il meccanismo della *paginazione* usato dalla memoria virtuale (vedi sez. 2.2.1) permette di *mappare* il contenuto di un file in una sezione dello spazio di indirizzi del processo che lo ha allocato.

Figura 10.15: Disposizione della memoria di un processo quando si esegue la mappatura in memoria di un file.

Il meccanismo è illustrato in fig. 10.15, una sezione del file viene *mappata* direttamente nello spazio degli indirizzi del programma. Tutte le operazioni di lettura e scrittura su variabili contenute in questa zona di memoria verranno eseguite leggendo e scrivendo dal contenuto del file attraverso il sistema della memoria virtuale illustrato in sez. 2.2.1 che in maniera analoga a quanto avviene per le pagine che vengono salvate e rilette nella *swap*, si incaricherà di sincronizzare il contenuto di quel segmento di memoria con quello del file mappato su di esso. Per questo motivo si può parlare tanto di *file mappato in memoria*, quanto di *memoria mappata su file*.

L'uso del *memory-mapping* comporta una notevole semplificazione delle operazioni di I/O, in quanto non sarà più necessario utilizzare dei buffer intermedi su cui appoggiare i dati da trasferire, poiché questi potranno essere acceduti direttamente nella sezione di memoria mappata; inoltre questa interfaccia è più efficiente delle usuali funzioni di I/O, in quanto permette di caricare in memoria solo le parti del file che sono effettivamente usate ad un dato istante.

Infatti, dato che l'accesso è fatto direttamente attraverso la memoria virtuale, la sezione di memoria mappata su cui si opera sarà a sua volta letta o scritta sul file una pagina alla volta e solo per le parti effettivamente usate, il tutto in maniera completamente trasparente al processo; l'accesso alle pagine non ancora caricate avverrà allo stesso modo con cui vengono caricate in memoria le pagine che sono state salvate sullo *swap*.

Infine in situazioni in cui la memoria è scarsa, le pagine che mappano un file vengono salvate automaticamente, così come le pagine dei programmi vengono scritte sulla *swap*; questo consente di accedere ai file su dimensioni il cui solo limite è quello dello spazio di indirizzi disponibile, e non della memoria su cui possono esserne lette delle porzioni.

L'interfaccia POSIX implementata da Linux prevede varie funzioni di sistema per la gestione del *memory mapped I/O*, la prima di queste, che serve ad eseguire la mappatura in memoria di un file, è `mmap`; il suo prototipo è:


```
#include <sys/mman.h>
void * mmap(void * start, size_t length, int prot, int flags, int fd, off_t offset)
    Esegue la mappatura in memoria di una sezione di un file.
```

La funzione ritorna il puntatore alla zona di memoria mappata in caso di successo, e MAP_FAILED ((void *) -1) per un errore, nel qual caso errno assumerà uno dei valori:

EACCES o fd non si riferisce ad un file regolare, o si è usato MAP_PRIVATE ma fd non è aperto in lettura, o si è usato MAP_SHARED e impostato PROT_WRITE ed fd non è aperto in lettura/scrittura, o si è impostato PROT_WRITE ed fd è in *append-only*.

EAGAIN il file è bloccato, o si è bloccata troppa memoria rispetto a quanto consentito dai limiti di sistema (vedi sez. 6.3.2).

EBADF il file descriptor non è valido, e non si è usato MAP_ANONYMOUS.

EINVAL i valori di start, length o offset non sono validi (o troppo grandi o non allineati sulla dimensione delle pagine), o length è zero (solo dal 2.6.12) o flags contiene sia MAP_PRIVATE che MAP_SHARED o nessuno dei due.

ENFILE si è superato il limite del sistema sul numero di file aperti (vedi sez. 6.3.2).

ENODEV il filesystem di fd non supporta il memory mapping.

ENOMEM non c'è memoria o si è superato il limite sul numero di mappature possibili.

EOVERFLOW su architettura a 32 bit con il supporto per i *large file* (che hanno una dimensione a 64 bit) il numero di pagine usato per length aggiunto a quello usato per offset eccede i 32 bit (unsigned long).

EPERM l'argomento prot ha richiesto PROT_EXEC, ma il filesystem di fd è montato con l'opzione noexec.

ETXTBSY si è impostato MAP_DENYWRITE ma fd è aperto in scrittura.

La funzione richiede di mappare in memoria la sezione del file fd a partire da offset per length byte, preferibilmente all'indirizzo start. Il valore start viene normalmente considerato come un suggerimento, ma l'uso di un qualunque valore diverso da NULL, in cui si rimette completamente al kernel la scelta dell'indirizzo, viene sconsigliato per ragioni di portabilità. Il valore di offset deve essere un multiplo della dimensione di una pagina di memoria.

Valore	Significato
PROT_EXEC	Le pagine possono essere eseguite.
PROT_READ	Le pagine possono essere lette.
PROT_WRITE	Le pagine possono essere scritte.
PROT_NONE	L'accesso alle pagine è vietato.

Tabella 10.14: Valori dell'argomento prot di mmap, relativi alla protezione applicate alle pagine del file mappate in memoria.

Il valore dell'argomento prot indica la protezione⁴⁷ da applicare al segmento di memoria e deve essere specificato come maschera binaria ottenuta dall'OR di uno o più dei valori riportati in tab. 10.14; il valore specificato deve essere compatibile con la modalità di accesso con cui si è aperto il file.

L'argomento flags specifica infine qual è il tipo di oggetto mappato, le opzioni relative alle modalità con cui è effettuata la mappatura e alle modalità con cui le modifiche alla memoria mappata vengono condivise o mantenute private al processo che le ha effettuate. Deve essere specificato come maschera binaria ottenuta dall'OR di uno o più dei valori riportati in

⁴⁷come accennato in sez. 2.2 in Linux la memoria reale è divisa in pagine, ogni processo vede la sua memoria attraverso uno o più segmenti lineari di memoria virtuale; per ciascuno di questi segmenti il kernel mantiene nella *page table* la mappatura sulle pagine di memoria reale, ed le modalità di accesso (lettura, esecuzione, scrittura); una loro violazione causa quella una *segment violation*, e la relativa emissione del segnale SIGSEGV.

Valore	Significato
MAP_32BIT	Esegue la mappatura sui primi 2Gb dello spazio degli indirizzi, viene supportato solo sulle piattaforme x86-64 per compatibilità con le applicazioni a 32 bit. Viene ignorato se si è richiesto MAP_FIXED (dal kernel 2.4.20).
MAP_ANON	Sinonimo di MAP_ANONYMOUS, deprecato.
MAP_ANONYMOUS	La mappatura non è associata a nessun file. Gli argomenti <code>fd</code> e <code>offset</code> sono ignorati. L'uso di questo flag con MAP_SHARED è stato implementato in Linux a partire dai kernel della serie 2.4.x.
MAP_DENYWRITE	In Linux viene ignorato per evitare <i>DoS</i> (veniva usato per segnalare che tentativi di scrittura sul file dovevano fallire con ETXTBSY).
MAP_EXECUTABLE	Ignorato.
MAP_FILE	Valore di compatibilità, ignorato.
MAP_FIXED	Non permette di restituire un indirizzo diverso da <code>start</code> , se questo non può essere usato <code>mmap</code> fallisce. Se si imposta questo flag il valore di <code>start</code> deve essere allineato alle dimensioni di una pagina.
MAP_GROWSDOWN	Usato per gli <i>stack</i> . Indica che la mappatura deve essere effettuata con gli indirizzi crescenti verso il basso.
MAP_HUGETLB	Esegue la mappatura usando le cosiddette “ <i>huge pages</i> ” (dal kernel 2.6.32).
MAP_LOCKED	Se impostato impedisce lo <i>swapping</i> delle pagine mappate (dal kernel 2.5.37).
MAP_NONBLOCK	Esegue un <i>prefaulting</i> più limitato che non causa I/O (dal kernel 2.5.46).
MAP_NORESERVE	Si usa con MAP_PRIVATE. Non riserva delle pagine di <i>swap</i> ad uso del meccanismo del <i>copy on write</i> per mantenere le modifiche fatte alla regione mappata, in questo caso dopo una scrittura, se non c'è più memoria disponibile, si ha l'emissione di un SIGSEGV.
MAP_POPULATE	Esegue il <i>prefaulting</i> delle pagine di memoria necessarie alla mappatura (dal kernel 2.5.46).
MAP_PRIVATE	I cambiamenti sulla memoria mappata non vengono riportati sul file. Ne viene fatta una copia privata cui solo il processo chiamante ha accesso. Incompatibile con MAP_SHARED.
MAP_SHARED	I cambiamenti sulla memoria mappata vengono riportati sul file e saranno immediatamente visibili agli altri processi che mappano lo stesso file. Incompatibile con MAP_PRIVATE.
MAP_STACK	Al momento è ignorato, è stato fornito (dal kernel 2.6.27) a supporto della implementazione dei <i>thread</i> nella <i>glibc</i> , per allocare memoria in uno spazio utilizzabile come <i>stack</i> per le architetture hardware che richiedono un trattamento speciale di quest'ultimo.
MAP_UNINITIALIZED	Specifico per i sistemi embedded ed utilizzabile dal kernel 2.6.33 solo se è stata abilitata in fase di compilazione dello stesso l'opzione CONFIG_MMMap_ALLOW_UNINITIALIZED. Se usato le pagine di memoria usate nella mappatura anonima non vengono cancellate; questo migliora le prestazioni sui sistemi con risorse minime, ma comporta la possibilità di rileggere i dati di altri processi che han chiuso una mappatura, per cui viene usato solo quando (come si suppone sia per i sistemi embedded) si ha il completo controllo dell'uso della memoria da parte degli utenti.

Tabella 10.15: Valori possibili dell'argomento `flag` di `mmap`.

tab. 10.15. Fra questi comunque deve sempre essere specificato o MAP_PRIVATE o MAP_SHARED per indicare la modalità con cui viene effettuata la mappatura.

Esistono infatti due modalità alternative di eseguire la mappatura di un file; la più comune è MAP_SHARED in cui la memoria è condivisa e le modifiche effettuate su di essa sono visibili a tutti i processi che hanno mappato lo stesso file. In questo caso le modifiche vengono anche riportate su disco, anche se questo può non essere immediato a causa della bufferizzazione: si potrà essere sicuri dell'aggiornamento solo in seguito alla chiamata di `msync` o `munmap`, e solo allora le modifiche saranno visibili sul file con l'I/O convenzionale.

Con MAP_PRIVATE invece viene creata una copia privata del file, questo non viene mai modificato e solo il processo chiamante ha accesso alla mappatura. Le modifiche eseguite dal processo sulla mappatura vengono effettuate utilizzando il meccanismo del *copy on write*,

mentenute in memoria e salvate su *swap* in caso di necessità. Non è specificato se i cambiamenti sul file originale vengano riportati sulla regione mappata.

Gli altri valori di `flag` modificano le caratteristiche della mappatura. Fra questi il più rilevante è probabilmente `MAP_ANONYMOUS` che consente di creare segmenti di memoria condivisa fra processi diversi senza appoggiarsi a nessun file (torneremo sul suo utilizzo in sez. 11.3.4). In tal caso gli argomenti `fd` e `offset` vengono ignorati, anche se alcune implementazioni richiedono che invece `fd` sia `-1`, convenzione che è opportuno seguire se si ha a cuore la portabilità dei programmi.

Gli effetti dell'accesso ad una zona di memoria mappata su file possono essere piuttosto complessi, essi si possono comprendere solo tenendo presente che tutto quanto è comunque basato sul meccanismo della memoria virtuale. Questo comporta allora una serie di conseguenze. La più ovvia è che se si cerca di scrivere su una zona mappata in sola lettura si avrà l'emissione di un segnale di violazione di accesso (`SIGSEGV`), dato che i permessi sul segmento di memoria relativo non consentono questo tipo di accesso.

È invece assai diversa la questione relativa agli accessi al di fuori della regione di cui si è richiesta la mappatura. A prima vista infatti si potrebbe ritenere che anch'essi debbano generare un segnale di violazione di accesso; questo però non tiene conto del fatto che, essendo basata sul meccanismo della paginazione, la mappatura in memoria non può che essere eseguita su un segmento di dimensioni rigorosamente multiple di quelle di una pagina, ed in generale queste potranno non corrispondere alle dimensioni effettive del file o della sezione che si vuole mappare.

Figura 10.16: Schema della mappatura in memoria di una sezione di file di dimensioni non corrispondenti al bordo di una pagina.

Il caso più comune è quello illustrato in fig. 10.16, in cui la sezione di file non rientra nei confini di una pagina: in tal caso il file sarà mappato su un segmento di memoria che si estende fino al bordo della pagina successiva. In questo caso è possibile accedere a quella zona di memoria che eccede le dimensioni specificate da `length`, senza ottenere un `SIGSEGV` poiché essa è presente nello spazio di indirizzi del processo, anche se non è mappata sul file. Il comportamento del sistema è quello di restituire un valore nullo per quanto viene letto, e di non riportare su file quanto viene scritto.

Un caso più complesso è quello che si viene a creare quando le dimensioni del file mappato sono più corte delle dimensioni della mappatura, oppure quando il file è stato troncato, dopo che è stato mappato, ad una dimensione inferiore a quella della mappatura in memoria. In questa situazione, per la sezione di pagina parzialmente coperta dal contenuto del file, vale esattamente quanto visto in precedenza; invece per la parte che eccede, fino alle dimensioni date da `length`, l'accesso non sarà più possibile, ma il segnale emesso non sarà `SIGSEGV`, ma `SIGBUS`, come illustrato in fig. 10.17.

Non tutti i file possono venire mappati in memoria, dato che, come illustrato in fig. 10.15, la mappatura introduce una corrispondenza biunivoca fra una sezione di un file ed una sezione di memoria. Questo comporta che ad esempio non è possibile mappare in memoria file descriptor relativi a *pipe*, *socket* e *fifo*, per i quali non ha senso parlare di *sezione*. Lo stesso vale anche per alcuni file di dispositivo, che non dispongono della relativa operazione `mmap` (si ricordi quanto esposto in sez. 4.1.1). Si tenga presente però che esistono anche casi di dispositivi (un esempio è l'interfaccia al ponte PCI-VME del chip Universe) che sono utilizzabili solo con questa interfaccia.

Dato che passando attraverso una *fork* lo spazio di indirizzi viene copiato integralmente, i file mappati in memoria verranno ereditati in maniera trasparente dal processo figlio,

Figura 10.17: Schema della mappatura in memoria di file di dimensioni inferiori alla lunghezza richiesta.

mantenendo gli stessi attributi avuti nel padre; così se si è usato `MAP_SHARED` padre e figlio accederanno allo stesso file in maniera condivisa, mentre se si è usato `MAP_PRIVATE` ciascuno di essi manterrà una sua versione privata indipendente. Non c'è invece nessun passaggio attraverso una `exec`, dato che quest'ultima sostituisce tutto lo spazio degli indirizzi di un processo con quello di un nuovo programma.

Quando si effettua la mappatura di un file vengono pure modificati i tempi ad esso associati (di cui si è trattato in sez. 4.3.4). Il valore di `st_atime` può venir cambiato in qualunque istante a partire dal momento in cui la mappatura è stata effettuata: il primo riferimento ad una pagina mappata su un file aggiorna questo tempo. I valori di `st_ctime` e `st_mtime` possono venir cambiati solo quando si è consentita la scrittura sul file (cioè per un file mappato con `PROT_WRITE` e `MAP_SHARED`) e sono aggiornati dopo la scrittura o in corrispondenza di una eventuale `msync`.

Dato per i file mappati in memoria le operazioni di I/O sono gestite direttamente dalla memoria virtuale, occorre essere consapevoli delle interazioni che possono esserci con operazioni effettuate con l'interfaccia dei file ordinaria illustrata in sez. 5.1. Il problema è che una volta che si è mappato un file, le operazioni di lettura e scrittura saranno eseguite sulla memoria, e riportate su disco in maniera autonoma dal sistema della memoria virtuale.

Pertanto se si modifica un file con l'interfaccia ordinaria queste modifiche potranno essere visibili o meno a seconda del momento in cui la memoria virtuale trasporterà dal disco in memoria quella sezione del file, perciò è del tutto imprevedibile il risultato della modifica di un file nei confronti del contenuto della memoria su cui è mappato.

Per questo è sempre sconsigliabile eseguire scritture su un file attraverso l'interfaccia ordinaria quando lo si è mappato in memoria, è invece possibile usare l'interfaccia ordinaria per leggere un file mappato in memoria, purché si abbia una certa cura; infatti l'interfaccia dell'I/O mappato in memoria mette a disposizione la funzione `msync` per sincronizzare il contenuto della memoria mappata con il file su disco; il suo prototipo è:

<pre>#include <sys/mman.h> int msync(const void *start, size_t length, int flags)</pre>	
Sincronizza i contenuti di una sezione di un file mappato in memoria.	
La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
<code>EBUSY</code>	si è indicato <code>MS_INVALIDATE</code> ma nell'intervallo di memoria specificato è presente un <i>memory lock</i> .
<code>EFAULT</code>	l'intervallo indicato, o parte di esso, non risulta mappato (prima del kernel 2.4.19).
<code>EINVAL</code>	o <code>start</code> non è multiplo di <code>PAGE_SIZE</code> , o si è specificato un valore non valido per <code>flags</code> .
<code>ENOMEM</code>	l'intervallo indicato, o parte di esso, non risulta mappato (dal kernel 2.4.19).

La funzione esegue la sincronizzazione di quanto scritto nella sezione di memoria indicata da `start` e `offset`, scrivendo le modifiche sul file (qualora questo non sia già stato fatto). Provvede anche ad aggiornare i relativi tempi di modifica. In questo modo si è sicuri che dopo l'esecuzione di `msync` le funzioni dell'interfaccia ordinaria troveranno un contenuto del file aggiornato.

L'argomento `flag` è specificato come maschera binaria composta da un OR dei valori riportati in tab. 10.16, di questi però `MS_ASYNC` e `MS_SYNC` sono incompatibili; con il primo valore infatti la funzione si limita ad inoltrare la richiesta di sincronizzazione al meccanismo della

Valore	Significato
MS_SYNC	richiede una sincronizzazione e ritorna soltanto quando questa è stata completata.
MS_ASYNC	richiede una sincronizzazione, ma ritorna subito non attendendo che questa sia finita.
MS_INVALIDATE	invalida le pagine per tutte le mappature in memoria così da rendere necessaria una riletture immediata delle stesse.

Tabella 10.16: Valori possibili dell'argomento `flag` di `msync`.

memoria virtuale, ritornando subito, mentre con il secondo attende che la sincronizzazione sia stata effettivamente eseguita. Il terzo valore fa sì che vengano invalidate, per tutte le mappature dello stesso file, le pagine di cui si è richiesta la sincronizzazione, così che esse possano essere immediatamente aggiornate con i nuovi valori.

Una volta che si sono completate le operazioni di I/O si può eliminare la mappatura della memoria usando la funzione `munmap`, il suo prototipo è:

```
#include <sys/mman.h>
int munmap(void *start, size_t length)
    Rilascia la mappatura sulla sezione di memoria specificata.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL l'intervallo specificato non ricade in una zona precedentemente mappata.

La funzione cancella la mappatura per l'intervallo specificato con `start` e `length`; ogni successivo accesso a tale regione causerà un errore di accesso in memoria. L'argomento `start` deve essere allineato alle dimensioni di una pagina, e la mappatura di tutte le pagine contenute anche parzialmente nell'intervallo indicato, verrà rimossa. Indicare un intervallo che non contiene mappature non è un errore. Si tenga presente inoltre che alla conclusione di un processo ogni pagina mappata verrà automaticamente rilasciata, mentre la chiusura del file descriptor usato per il *memory mapping* non ha alcun effetto su di esso.

Lo standard POSIX prevede anche una funzione che permetta di cambiare le protezioni delle pagine di memoria; lo standard prevede che essa si applichi solo ai *memory mapping* creati con `mmap`, ma nel caso di Linux la funzione può essere usata con qualunque pagina valida nella memoria virtuale. Questa funzione di sistema è `mprotect` ed il suo prototipo è:

```
#include <sys/mman.h>
int mprotect(const void *addr, size_t len, int prot)
    Modifica le protezioni delle pagine di memoria.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL il valore di `addr` non è valido o non è un multiplo di `PAGE_SIZE`.

EACCES l'operazione non è consentita, ad esempio si è cercato di marcare con `PROT_WRITE` un segmento di memoria cui si ha solo accesso in lettura.

ENOMEM non è stato possibile allocare le risorse necessarie all'interno del kernel o si è specificato un indirizzo di memoria non valido del processo o non corrispondente a pagine mappate (negli ultimi due casi prima del kernel 2.4.19 veniva prodotto, erroneamente, `EFAULT`).

La funzione prende come argomenti un indirizzo di partenza in `addr`, allineato alle dimensioni delle pagine di memoria, ed una dimensione `size`. La nuova protezione deve essere specificata in `prot` con una combinazione dei valori di tab. 10.14. La nuova protezione verrà applicata a tutte le pagine contenute, anche parzialmente, dall'intervallo fra `addr` e `addr+size-1`.

Infine Linux supporta alcune operazioni specifiche non disponibili su altri kernel unix-like per poter usare le quali occorre però dichiarare `_GNU_SOURCE` prima dell'inclusione di `sys/mman.h`. La prima di queste è la possibilità di modificare un precedente *memory mapping*, ad esempio per espanderlo o restringerlo. Questo è realizzato dalla funzione di sistema `mremap`, il cui prototipo è:

```
#include <sys/mman.h>
void * mremap(void *old_address, size_t old_size , size_t new_size, unsigned long flags)
```

Restringe o allarga una mappatura in memoria.

La funzione ritorna l'indirizzo alla nuova area di memoria in caso di successo o il valore `MAP_FAILED` (pari a `(void *) -1`), nel qual caso `errno` assumerà uno dei valori:

<code>EINVAL</code>	il valore di <code>old_address</code> non è un puntatore valido.
<code>EFAULT</code>	ci sono indirizzi non validi nell'intervallo specificato da <code>old_address</code> e <code>old_size</code> , o ci sono altre mappature di tipo non corrispondente a quella richiesta.
<code>ENOMEM</code>	non c'è memoria sufficiente oppure l'area di memoria non può essere espansa all'indirizzo virtuale corrente, e non si è specificato <code>MREMAP_MAYMOVE</code> nei flag.
<code>EAGAIN</code>	il segmento di memoria scelto è bloccato e non può essere rimappato.

La funzione richiede come argomenti `old_address` (che deve essere allineato alle dimensioni di una pagina di memoria) che specifica il precedente indirizzo del *memory mapping* e `old_size`, che ne indica la dimensione. Con `new_size` si specifica invece la nuova dimensione che si vuole ottenere. Infine l'argomento `flags` è una maschera binaria per i flag che controllano il comportamento della funzione. Il solo valore utilizzato è `MREMAP_MAYMOVE` che consente di eseguire l'espansione anche quando non è possibile utilizzare il precedente indirizzo. Per questo motivo, se si è usato questo flag, la funzione può restituire un indirizzo della nuova zona di memoria che non è detto coincida con `old_address`.

La funzione si appoggia al sistema della memoria virtuale per modificare l'associazione fra gli indirizzi virtuali del processo e le pagine di memoria, modificando i dati direttamente nella *page table* del processo. Come per `mprotect` la funzione può essere usata in generale, anche per pagine di memoria non corrispondenti ad un *memory mapping*, e consente così di implementare la funzione `realloc` in maniera molto efficiente.

Una caratteristica comune a tutti i sistemi unix-like è che la mappatura in memoria di un file viene eseguita in maniera lineare, cioè parti successive di un file vengono mappate linearmente su indirizzi successivi in memoria. Esistono però delle applicazioni (in particolare la tecnica è usata dai database o dai programmi che realizzano macchine virtuali) in cui è utile poter mappare sezioni diverse di un file su diverse zone di memoria.

Questo è ovviamente sempre possibile eseguendo ripetutamente la funzione `mmap` per ciascuna delle diverse aree del file che si vogliono mappare in sequenza non lineare (ed in effetti è quello che veniva fatto anche con Linux prima che fossero introdotte queste estensioni) ma questo approccio ha delle conseguenze molto pesanti in termini di prestazioni. Infatti per ciascuna mappatura in memoria deve essere definita nella *page table* del processo una nuova area di memoria virtuale, quella che nel gergo del kernel viene chiamata VMA (*virtual memory area*, che corrisponda alla mappatura, in modo che questa diventi visibile nello spazio degli indirizzi come illustrato in fig. 10.15.

Quando un processo esegue un gran numero di mappature diverse (si può arrivare anche a centinaia di migliaia) per realizzare a mano una mappatura non-lineare esso vedrà un accrescimento eccessivo della sua *page table*, e lo stesso accadrà per tutti gli altri processi che utilizzano questa tecnica. In situazioni in cui le applicazioni hanno queste esigenze si avranno delle prestazioni ridotte, dato che il kernel dovrà impiegare molte risorse per mantenere i dati

relativi al *memory mapping*, sia in termini di memoria interna per i dati delle *page table*, che di CPU per il loro aggiornamento.

Per questo motivo con il kernel 2.5.46 è stato introdotto, ad opera di Ingo Molnar, un meccanismo che consente la mappatura non-lineare. Anche questa è una caratteristica specifica di Linux, non presente in altri sistemi unix-like. Diventa così possibile utilizzare una sola mappatura iniziale, e quindi una sola *virtual memory area* nella *page table* del processo, e poi rimappare a piacere all'interno di questa i dati del file. Ciò è possibile grazie ad una nuova *system call*, `remap_file_pages`, il cui prototipo è:

```
#include <sys/mman.h>
int remap_file_pages(void *start, size_t size, int prot, ssize_t pgoff, int flags)
                                Rimappa non linearmente un memory mapping.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` si è usato un valore non valido per uno degli argomenti o `start` non fa riferimento ad un *memory mapping* valido creato con `MAP_SHARED`.
ed inoltre nel loro significato generico.

Per poter utilizzare questa funzione occorre anzitutto effettuare preliminarmente una chiamata a `mmap` con `MAP_SHARED` per definire l'area di memoria che poi sarà rimappata non linearmente. Poi si chiamerà questa funzione per modificare le corrispondenze fra pagine di memoria e pagine del file; si tenga presente che `remap_file_pages` permette anche di mappare la stessa pagina di un file in più pagine della regione mappata.

La funzione richiede che si identifichi la sezione del file che si vuole riposizionare all'interno del *memory mapping* con gli argomenti `pgoff` e `size`; l'argomento `start` invece deve indicare un indirizzo all'interno dell'area definita dall'`mmap` iniziale, a partire dal quale la sezione di file indicata verrà rimappata. L'argomento `prot` deve essere sempre nullo, mentre `flags` prende gli stessi valori di `mmap` (quelli di tab. 10.14) ma di tutti i flag solo `MAP_NONBLOCK` non viene ignorato.

Insieme alla funzione `remap_file_pages` nel kernel 2.5.46 con sono stati introdotti anche due nuovi flag per `mmap`: `MAP_POPULATE` e `MAP_NONBLOCK`. Il primo dei due consente di abilitare il meccanismo del *prefaulting*. Questo viene di nuovo in aiuto per migliorare le prestazioni in certe condizioni di utilizzo del *memory mapping*.

Il problema si pone tutte le volte che si vuole mappare in memoria un file di grosse dimensioni. Il comportamento normale del sistema della memoria virtuale è quello per cui la regione mappata viene aggiunta alla *page table* del processo, ma i dati verranno effettivamente utilizzati (si avrà cioè un *page fault* che li trasferisce dal disco alla memoria) soltanto in corrispondenza dell'accesso a ciascuna delle pagine interessate dal *memory mapping*.

Questo vuol dire che il passaggio dei dati dal disco alla memoria avverrà una pagina alla volta con un gran numero di *page fault*, chiaramente se si sa in anticipo che il file verrà utilizzato immediatamente, è molto più efficiente eseguire un *prefaulting* in cui tutte le pagine di memoria interessate alla mappatura vengono “popolate” in una sola volta, questo comportamento viene abilitato quando si usa con `mmap` il flag `MAP_POPULATE`.

Dato che l'uso di `MAP_POPULATE` comporta dell'I/O su disco che può rallentare l'esecuzione di `mmap` è stato introdotto anche un secondo flag, `MAP_NONBLOCK`, che esegue un *prefaulting* più limitato in cui vengono popolate solo le pagine della mappatura che già si trovano nella cache del kernel.⁴⁸

Per i vantaggi illustrati all'inizio del paragrafo l'interfaccia del *memory mapped I/O* viene usata da una grande varietà di programmi, spesso con esigenze molto diverse fra di loro

⁴⁸questo può essere utile per il linker dinamico, in particolare quando viene effettuato il *prelink* delle applicazioni.

riguardo le modalità con cui verranno eseguiti gli accessi ad un file; è ad esempio molto comune per i database effettuare accessi ai dati in maniera pressoché casuale, mentre un riproduttore audio o video eseguirà per lo più letture sequenziali.

Per migliorare le prestazioni a seconda di queste modalità di accesso è disponibile una apposita funzione, `madvise`,⁴⁹ che consente di fornire al kernel delle indicazioni su come un processo intende accedere ad un segmento di memoria, anche al di là delle mappature dei file, così che possano essere adottate le opportune strategie di ottimizzazione. Il suo prototipo è:

```
#include <sys/mman.h>
int madvise(void *start, size_t length, int advice)
    Fornisce indicazioni sull'uso previsto di un segmento di memoria.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF	la mappatura esiste ma non corrisponde ad un file.
EINVAL	<code>start</code> non è allineato alla dimensione di una pagina, <code>length</code> ha un valore negativo, o <code>advice</code> non è un valore valido, o si è richiesto il rilascio (con <code>MADV_DONTNEED</code>) di pagine bloccate o condivise o si è usato <code>MADV_MERGEABLE</code> o <code>MADV_UNMERGEABLE</code> ma il kernel non è stato compilato per il relativo supporto.
EIO	la paginazione richiesta eccederebbe i limiti (vedi sez. 6.3.2) sulle pagine residenti in memoria del processo (solo in caso di <code>MADV_WILLNEED</code>).
ENOMEM	gli indirizzi specificati non sono mappati, o, in caso <code>MADV_WILLNEED</code> , non c'è sufficiente memoria per soddisfare la richiesta.

ed inoltre `EAGAIN` e `ENOSYS` nel loro significato generico.

La sezione di memoria sulla quale si intendono fornire le indicazioni deve essere indicata con l'indirizzo iniziale `start` e l'estensione `length`, il valore di `start` deve essere allineato, mentre `length` deve essere un numero positivo; la versione di Linux consente anche un valore nullo per `length`, inoltre se una parte dell'intervallo non è mappato in memoria l'indicazione viene comunque applicata alle restanti parti, anche se la funzione ritorna un errore di `ENOMEM`.

L'indicazione viene espressa dall'argomento `advice` che deve essere specificato con uno dei valori riportati in tab. 10.17; si tenga presente che i valori indicati nella seconda parte della tabella sono specifici di Linux e non sono previsti dallo standard POSIX.1b. La funzione non ha, tranne il caso di `MADV_DONTFORK`, nessun effetto sul comportamento di un programma, ma può influenzarne le prestazioni fornendo al kernel indicazioni sulle esigenze dello stesso, così che sia possibile scegliere le opportune strategie per la gestione del *read-ahead* (vedi sez. 10.4.4) e del caching dei dati.

A differenza da quanto specificato nello standard POSIX.1b, per il quale l'uso di `madvise` è a scopo puramente indicativo, Linux considera queste richieste come imperative, per cui ritorna un errore qualora non possa soddisfarle; questo comportamento differisce da quanto specificato nello standard.

Nello standard POSIX.1-2001 è prevista una ulteriore funzione `posix_madvise` che su Linux viene reimplementata utilizzando `madvise`; il suo prototipo è:

⁴⁹tratteremo in sez. 10.4.4 le funzioni che consentono di ottimizzare l'accesso ai file con l'interfaccia classica.

⁵⁰a partire dal kernel 2.6.32 è stato introdotto un meccanismo che identifica pagine di memoria identiche e le accorpa in una unica pagina (soggetta al *copy-on-write* per successive modifiche); per evitare di controllare tutte le pagine solo quelle marcate con questo flag vengono prese in considerazione per l'accorpamento; in questo modo si possono migliorare le prestazioni nella gestione delle macchine virtuali diminuendo la loro occupazione di memoria, ma il meccanismo può essere usato anche in altre applicazioni in cui sian presenti numerosi processi che usano gli stessi dati; per maggiori dettagli si veda http://kernelnewbies.org/Linux_2_6_32 e la documentazione nei sorgenti del kernel (Documentation/vm/ksm.txt).

Valore	Significato
MADV_DONTNEED	non ci si aspetta nessun accesso nell'immediato futuro, pertanto le pagine possono essere liberate dal kernel non appena necessario; l'area di memoria resterà accessibile, ma un accesso richiederà che i dati vengano ricaricati dal file a cui la mappatura fa riferimento.
MADV_NORMAL	nessuna indicazione specifica, questo è il valore di default usato quando non si è chiamato <code>madvise</code> .
MADV_RANDOM	ci si aspetta un accesso casuale all'area indicata, pertanto l'applicazione di una lettura anticipata con il meccanismo del <i>read-ahead</i> (vedi sez. 10.4.4) è di scarsa utilità e verrà disabilitata.
MADV_SEQUENTIAL	ci si aspetta un accesso sequenziale al file, quindi da una parte sarà opportuno eseguire una lettura anticipata, e dall'altra si potranno scartare immediatamente le pagine una volta che queste siano state lette.
MADV_WILLNEED	ci si aspetta un accesso nell'immediato futuro, pertanto l'applicazione del <i>read-ahead</i> deve essere incentivata.
MADV_DONTDUMP	esclude da un <i>core dump</i> (vedi sez. 7.2.1) le pagine specificate, viene usato per evitare di scrivere su disco dati relativi a zone di memoria che si sa non essere utili in un <i>core dump</i> .
MADV_DODUMP	rimuove l'effetto della precedente MADV_DONTDUMP (dal kernel 3.4).
MADV_DONTFORK	impedisce che l'intervallo specificato venga ereditato dal processo figlio dopo una fork; questo consente di evitare che il meccanismo del <i>copy on write</i> effettui la rilocalizzazione delle pagine quando il padre scrive sull'area di memoria dopo la fork, cosa che può causare problemi per l'hardware che esegue operazioni in DMA su quelle pagine (dal kernel 2.6.16).
MADV_DOFORK	rimuove l'effetto della precedente MADV_DONTFORK (dal kernel 2.6.16).
MADV_HUGEPAGE	abilita il meccanismo delle <i>Transparent Huge Page</i> (vedi sez. ??) sulla regione indicata; se questa è allineata alle relative dimensioni il kernel alloca direttamente delle <i>huge page</i> ; è utilizzabile solo con mappature anonime private (dal kernel 2.6.38).
MADV_NOHUGEPAGE	impedisce che la regione indicata venga collassata in eventuali <i>huge page</i> (dal kernel 2.6.38).
MADV_HWPOISON	opzione ad uso di debug per verificare codice che debba gestire errori nella gestione della memoria; richiede una apposita opzione di compilazione del kernel, privilegi amministrativi (la capacità CAP_SYS_ADMIN) e provoca l'emissione di un segnale di SIGBUS dal programma chiamante e rimozione della mappatura (dal kernel 2.6.32).
MADV_SOFT_OFFLINE	opzione utilizzata per il debug del codice di verifica degli errori di gestione memoria, richiede una apposita opzione di compilazione (dal kernel 2.6.33).
MADV_MERGEABLE	marca la pagina come accorpabile, indicazione principalmente ad uso dei sistemi di virtualizzazione ⁵⁰ (dal kernel 2.6.32).
MADV_REMOVE	libera un intervallo di pagine di memoria ed il relativo supporto sottostante; è supportato soltanto sui filesystem in RAM <i>tmpfs</i> e <i>shmfs</i> se usato su altri tipi di filesystem causa un errore di ENOSYS (dal kernel 2.6.16).
MADV_UNMERGEABLE	rimuove l'effetto della precedente MADV_MERGEABLE (dal kernel 2.6.32).

Tabella 10.17: Valori dell'argomento `advice` di `madvise`.

```
#include <sys/mman.h>
int posix_madvise(void *start, size_t length, int advice)
    Fornisce indicazioni sull'uso previsto di un segmento di memoria.
```

La funzione ritorna 0 in caso di successo ed un valore positivo per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL `start` non è allineato alla dimensione di una pagina, `length` ha un valore negativo, o `advice` non è un valore valido.

ENOMEM gli indirizzi specificati non sono nello spazio di indirizzi del processo.

Gli argomenti `start` e `length` hanno lo stesso identico significato degli analoghi di `madvise`, a cui si rimanda per la loro descrizione ma a differenza di quanto indicato dallo standard per

questa funzione, su Linux un valore nullo di `len` è consentito.

Valore	Significato
POSIX_MADV_DONTNEED	analogo a MADV_DONTNEED.
POSIX_MADV_NORMAL	identico a MADV_NORMAL.
POSIX_MADV_RANDOM	identico a MADV_RANDOM.
POSIX_MADV_SEQUENTIAL	identico a MADV_SEQUENTIAL.
POSIX_MADV_WILLNEED	identico a MADV_WILLNEED.

Tabella 10.18: Valori dell'argomento `advise` di `posix_madvise`.

L'argomento `advise` invece può assumere solo i valori indicati in tab. 10.18, che riflettono gli analoghi di `madvise`, con lo stesso effetto per tutti tranne `POSIX_MADV_DONTNEED`. Infatti a partire dalla *glibc* 2.6 `POSIX_MADV_DONTNEED` viene ignorato, in quanto l'uso del corrispondente `MADV_DONTNEED` di `madvise` ha, per la semantica imperativa, l'effetto immediato di far liberare le pagine da parte del kernel, che viene considerato distruttivo.

10.4.2 I/O vettorizzato: `readv` e `writev`

Una seconda modalità di I/O diversa da quella ordinaria è il cosiddetto *I/O vettorizzato*, che nasce per rispondere al caso abbastanza comune in cui ci si trova nell'esigenza di dover eseguire una serie multipla di operazioni di I/O, come una serie di letture o scritture di vari buffer. Un esempio tipico è quando i dati sono strutturati nei campi di una struttura ed essi devono essere caricati o salvati su un file. Benché l'operazione sia facilmente eseguibile attraverso una serie multipla di chiamate a `read` e `write`, ci sono casi in cui si vuole poter contare sulla atomicità delle operazioni di lettura e scrittura rispetto all'esecuzione del programma.

Per questo motivo fino da BSD 4.2 vennero introdotte delle nuove *system call* che permettessero di effettuare con una sola chiamata una serie di letture da, o scritture su, una serie di buffer, quello che poi venne chiamato *I/O vettorizzato*. Queste funzioni di sistema sono `readv` e `writev`,⁵¹ ed i relativi prototipi sono:

```
#include <sys/uio.h>
int readv(int fd, const struct iovec *vector, int count)
int writev(int fd, const struct iovec *vector, int count)
```

Eseguono rispettivamente una lettura o una scrittura vettorizzata.

Le funzioni ritornano il numero di byte letti o scritti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` si è specificato un valore non valido per uno degli argomenti (ad esempio `count` è maggiore di `IOV_MAX`).

più tutti i valori, con lo stesso significato, che possono risultare dalle condizioni di errore di `read` e `write`.

Entrambe le funzioni usano una struttura `iovec`, la cui definizione è riportata in fig. 10.18, che definisce dove i dati devono essere letti o scritti ed in che quantità. Il primo campo della struttura, `iov_base`, contiene l'indirizzo del buffer ed il secondo, `iov_len`, la dimensione dello stesso.

La lista dei buffer da utilizzare viene indicata attraverso l'argomento `vector` che è un vettore di strutture `iovec`, la cui lunghezza è specificata dall'argomento `count`.⁵² Ciascuna struttura dovrà essere inizializzata opportunamente per indicare i vari buffer da e verso i quali

⁵¹in Linux le due funzioni sono riprese da BSD4.4, esse sono previste anche dallo standard POSIX.1-2001.

⁵²fino alle *libc5*, Linux usava `size_t` come tipo dell'argomento `count`, una scelta logica, che però è stata dismessa per restare aderenti allo standard POSIX.1-2001.

```

struct iovec {
    void *iov_base;      /* Starting address */
    size_t iov_len;      /* Length in bytes */
};

```

Figura 10.18: La struttura `iovec`, usata dalle operazioni di I/O vettorizzato.

verrà eseguito il trasferimento dei dati. Essi verranno letti (o scritti) nell'ordine in cui li si sono specificati nel vettore `vector`.

La standardizzazione delle due funzioni all'interno della revisione POSIX.1-2001 prevede anche che sia possibile avere un limite al numero di elementi del vettore `vector`. Qualora questo sussista, esso deve essere indicato dal valore dalla costante `IOV_MAX`, definita come le altre costanti analoghe (vedi sez. 6.1.1) in `limits.h`; lo stesso valore deve essere ottenibile in esecuzione tramite la funzione `sysconf` richiedendo l'argomento `_SC_IOV_MAX` (vedi sez. 6.1.1).

Nel caso di Linux il limite di sistema è di 1024, però se si usa la *glibc* essa fornisce un *wrapper* per le *system call* che si accorge se una operazione supererà il precedente limite, in tal caso i dati verranno letti o scritti con le usuali `read` e `write` usando un buffer di dimensioni sufficienti appositamente allocato in grado di contenere tutti i dati indicati da `vector`. L'operazione avrà successo ma si perderà l'atomicità del trasferimento da e verso la destinazione finale.

Si tenga presente infine che queste funzioni operano sui file con l'interfaccia dei file descriptor, e non è consigliabile mescolarle con l'interfaccia classica dei *file stream* di sez. 5.3; a causa delle bufferizzazioni interne di quest'ultima infatti si potrebbero avere risultati indefiniti e non corrispondenti a quanto aspettato.

Come per le normali operazioni di lettura e scrittura, anche per l'I/O vettorizzato si pone il problema di poter effettuare le operazioni in maniera atomica a partire da una certa posizione sul file. Per questo motivo a partire dal kernel 2.6.30 sono state introdotte anche per l'I/O vettorizzato le analoghe delle funzioni `pread` e `pwrite` (vedi sez. 5.1.4 e 5.1.5); le due funzioni sono `preadv` e `pwritev` ed i rispettivi prototipi sono:⁵³

```

#include <sys/uio.h>
int preadv(int fd, const struct iovec *vector, int count, off_t offset)
int pwritev(int fd, const struct iovec *vector, int count, off_t offset)

```

Eseguono una lettura o una scrittura vettorizzata a partire da una data posizione sul file.

Le funzioni hanno gli stessi valori di ritorno delle corrispondenti `readv` e `writev` ed anche gli eventuali errori sono gli stessi, con in più quelli che si possono ottenere dalle possibili condizioni di errore di `lseek`.

Le due funzioni eseguono rispettivamente una lettura o una scrittura vettorizzata a partire dalla posizione `offset` sul file indicato da `fd`, la posizione corrente sul file, come vista da eventuali altri processi che vi facciano riferimento, non viene alterata. A parte la presenza dell'ulteriore argomento il comportamento delle funzioni è identico alle precedenti `readv` e `writev`.

Con l'uso di queste funzioni si possono evitare eventuali *race condition* quando si deve eseguire la una operazione di lettura e scrittura vettorizzata a partire da una certa posizione

⁵³Le due funzioni sono analoghe alle omonime presenti in BSD; le *system call* usate da Linux (introdotte a partire dalla versione 2.6.30) utilizzano degli argomenti diversi per problemi collegati al formato a 64 bit dell'argomento `offset`, che varia a seconda delle architetture, ma queste differenze vengono gestite dalle funzioni di librerie di libreria che mantengono l'interfaccia delle analoghe tratte da BSD.

su un file, mentre al contempo si possono avere in concorrenza processi che utilizzano lo stesso file descriptor (si ricordi quanto visto in sez. 5.2) con delle chiamate a `lseek`.

10.4.3 L'I/O diretto fra file descriptor: `sendfile` e `splice`

Uno dei problemi che si presentano nella gestione dell'I/O è quello in cui si devono trasferire grandi quantità di dati da un file descriptor ed un altro; questo usualmente comporta la lettura dei dati dal primo file descriptor in un buffer in memoria, da cui essi vengono poi scritti sul secondo.

Benché il kernel ottimizzi la gestione di questo processo quando si ha a che fare con file normali, in generale quando i dati da trasferire sono molti si pone il problema di effettuare trasferimenti di grandi quantità di dati da *kernel space* a *user space* e all'indietro, quando in realtà potrebbe essere più efficiente mantenere tutto in *kernel space*. Tratteremo in questa sezione alcune funzioni specialistiche che permettono di ottimizzare le prestazioni in questo tipo di situazioni.

La prima funzione che è stata ideata per ottimizzare il trasferimento dei dati fra due file descriptor è `sendfile`.⁵⁴ La funzione è presente in diverse versioni di Unix (la si ritrova ad esempio in FreeBSD, HPUNIX ed altri Unix) ma non è presente né in POSIX.1-2001 né in altri standard (pertanto si eviti di utilizzarla se si devono scrivere programmi portabili) per cui per essa vengono utilizzati prototipi e semantiche differenti. Nel caso di Linux il prototipo di `sendfile` è:

```
#include <sys/sendfile.h>
ssize_t sendfile(int out_fd, int in_fd, off_t *offset, size_t count)
```

Copia dei dati da un file descriptor ad un altro.

La funzione ritorna il numero di byte trasferiti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` si è impostata la modalità non bloccante su `out_fd` e la scrittura si bloccherebbe.

`EINVAL` i file descriptor non sono validi, o sono bloccati (vedi sez. 10.1), o `mmap` non è disponibile per `in_fd`.

`EIO` si è avuto un errore di lettura da `in_fd`.

`ENOMEM` non c'è memoria sufficiente per la lettura da `in_fd`.

ed inoltre `EBADF` e `EFAULT` nel loro significato generico.

La funzione copia direttamente `count` byte dal file descriptor `in_fd` al file descriptor `out_fd`. In caso di successo la funzione ritorna il numero di byte effettivamente copiati da `in_fd` a `out_fd` e come per le ordinarie `read` e `write` questo valore può essere inferiore a quanto richiesto con `count`.

Se il puntatore `offset` è nullo la funzione legge i dati a partire dalla posizione corrente su `in_fd`, altrimenti verrà usata la posizione indicata dal valore puntato da `offset`; in questo caso detto valore sarà aggiornato, come *value result argument*, per indicare la posizione del byte successivo all'ultimo che è stato letto, mentre la posizione corrente sul file non sarà modificata. Se invece `offset` è nullo la posizione corrente sul file sarà aggiornata tenendo conto dei byte letti da `in_fd`.

Fino ai kernel della serie 2.4 la funzione era utilizzabile su un qualunque file descriptor, e permetteva di sostituire la invocazione successiva di una `read` e una `write` (e l'allocazione del relativo buffer) con una sola chiamata a `sendfile`. In questo modo si poteva diminuire il numero di chiamate al sistema e risparmiare in trasferimenti di dati da *kernel space* a *user space* e viceversa. La massima utilità della funzione si ottiene comunque per il trasferimento

⁵⁴la funzione è stata introdotta con i kernel della serie 2.2, e disponibile dalla *glibc* 2.1.

di dati da un file su disco ad un socket di rete,⁵⁵ dato che in questo caso diventa possibile effettuare il trasferimento diretto via DMA dal controller del disco alla scheda di rete, senza neanche allocare un buffer nel kernel (il meccanismo è detto *zerocopy* in quanto i dati non vengono mai copiati dal kernel, che si limita a programmare solo le operazioni di lettura e scrittura via DMA) ottenendo la massima efficienza possibile senza pesare neanche sul processore.

In seguito però ci si accorse che, fatta eccezione per il trasferimento diretto da file a socket, non sempre `sendfile` comportava miglioramenti significativi delle prestazioni rispetto all'uso in sequenza di `read` e `write`. Nel caso generico infatti il kernel deve comunque allocare un buffer ed effettuare la copia dei dati, e in tal caso spesso il guadagno ottenibile nel ridurre il numero di chiamate al sistema non compensa le ottimizzazioni che possono essere fatte da una applicazione in *user space* che ha una conoscenza diretta su come questi sono strutturati, per cui in certi casi si potevano avere anche dei peggioramenti. Questo ha portato, per i kernel della serie 2.6,⁵⁶ alla decisione di consentire l'uso della funzione soltanto quando il file da cui si legge supporta le operazioni di *memory mapping* (vale a dire non è un socket) e quello su cui si scrive è un socket; in tutti gli altri casi l'uso di `sendfile` dà luogo ad un errore di `EINVAL`.

Nonostante ci possano essere casi in cui `sendfile` non migliora le prestazioni, resta il dubbio se la scelta di disabilitarla sempre per il trasferimento fra file di dati sia davvero corretta. Se ci sono peggioramenti di prestazioni infatti si può sempre fare ricorso al metodo ordinario, ma lasciare a disposizione la funzione consentirebbe se non altro di semplificare la gestione della copia dei dati fra file, evitando di dover gestire l'allocazione di un buffer temporaneo per il loro trasferimento. Comunque a partire dal kernel 2.6.33 la restrizione su `out_fd` è stata rimossa e questo può essere un file qualunque, rimane però quella di non poter usare un socket per `in_fd`.

A partire dal kernel 2.6.17 come alternativa a `sendfile` è disponibile la nuova *system call* `splice`. Lo scopo di questa funzione è quello di fornire un meccanismo generico per il trasferimento di dati da o verso un file, utilizzando un buffer gestito internamente dal kernel. Descritta in questi termini `splice` sembra semplicemente un “*dimezzamento*” di `sendfile`, nel senso che un trasferimento di dati fra due file con `sendfile` non sarebbe altro che la lettura degli stessi su un buffer seguita dalla relativa scrittura, cosa che in questo caso si dovrebbe eseguire con due chiamate a `splice`.

In realtà le due *system call* sono profondamente diverse nel loro meccanismo di funzionamento;⁵⁷ `sendfile` infatti, come accennato, non necessita di avere a disposizione un buffer interno, perché esegue un trasferimento diretto di dati; questo la rende in generale più efficiente, ma anche limitata nelle sue applicazioni, dato che questo tipo di trasferimento è possibile solo in casi specifici che nel caso di Linux questi sono anche solo quelli in cui essa può essere effettivamente utilizzata.

Il concetto che sta dietro a `splice` invece è diverso,⁵⁸ si tratta semplicemente di una funzione che consente di fare in maniera del tutto generica delle operazioni di trasferimento di dati fra un file e un buffer gestito interamente in *kernel space*. In questo caso il cuore

⁵⁵questo è il caso classico del lavoro eseguito da un server web, ed infatti Apache ha una opzione per il supporto esplicito di questa funzione.

⁵⁶per alcune motivazioni di questa scelta si può fare riferimento a quanto illustrato da Linus Torvalds in <http://www.cs.helsinki.fi/linux/linux-kernel/2001-03/0200.html>.

⁵⁷questo fino al kernel 2.6.23, dove `sendfile` è stata reimplementata in termini di `splice`, pur mantenendo disponibile la stessa interfaccia verso l'*user space*.

⁵⁸in realtà la proposta originale di Larry Mc Voy non differisce poi tanto negli scopi da `sendfile`, quello che rende `splice` davvero diversa è stata la reinterpretazione che ne è stata fatta nell'implementazione su Linux realizzata da Jens Anxboe, concetti che sono esposti sinteticamente dallo stesso Linus Torvalds in <http://kerneltrap.org/node/6505>.

della funzione (e delle affini `vmsplice` e `tee`, che tratteremo più avanti) è appunto l'uso di un buffer in *kernel space*, e questo è anche quello che ne ha semplificato l'adozione, perché l'infrastruttura per la gestione di un tale buffer è presente fin dagli albori di Unix per la realizzazione delle *pipe* (vedi sez. 11.1). Dal punto di vista concettuale allora `splice` non è altro che una diversa interfaccia (rispetto alle *pipe*) con cui utilizzare in *user space* l'oggetto “*buffer in kernel space*”.

Così se per una *pipe* o una *fifo* il buffer viene utilizzato come area di memoria (vedi fig. 11.1) dove appoggiare i dati che vengono trasferiti da un capo all'altro della stessa per creare un meccanismo di comunicazione fra processi, nel caso di `splice` il buffer viene usato o come fonte dei dati che saranno scritti su un file, o come destinazione dei dati che vengono letti da un file. La funzione fornisce quindi una interfaccia generica che consente di trasferire dati da un buffer ad un file o viceversa; il prototipo di `splice`, accessibile solo dopo aver definito la macro `_GNU_SOURCE`,⁵⁹ è il seguente:

<pre>#include <fcntl.h> long splice(int fd_in, off_t *off_in, int fd_out, off_t *off_out, size_t len, unsigned int flags)</pre>		Trasferisce dati da un file verso una <i>pipe</i> o viceversa.
La funzione ritorna il numero di byte trasferiti in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:		
EBADF	uno o entrambi fra <code>fd_in</code> e <code>fd_out</code> non sono file descriptor validi o, rispettivamente, non sono stati aperti in lettura o scrittura.	
EINVAL	il filesystem su cui si opera non supporta <code>splice</code> , oppure nessuno dei file descriptor è una <i>pipe</i> , oppure si è dato un valore a <code>off_in</code> o <code>off_out</code> ma il corrispondente file è un dispositivo che non supporta la funzione <code>lseek</code> .	
ENOMEM	non c'è memoria sufficiente per l'operazione richiesta.	
ESPIPE	o <code>off_in</code> o <code>off_out</code> non sono NULL ma il corrispondente file descriptor è una <i>pipe</i> .	

La funzione esegue un trasferimento di `len` byte dal file descriptor `fd_in` al file descriptor `fd_out`, uno dei quali deve essere una *pipe*; l'altro file descriptor può essere qualunque, questo significa che può essere, oltre che un file di dati, anche un'altra *pipe*, o un socket. Come accennato una *pipe* non è altro che un buffer in *kernel space*, per cui a seconda che essa sia usata per `fd_in` o `fd_out` si avrà rispettivamente la copia dei dati dal buffer al file o viceversa.

In caso di successo la funzione ritorna il numero di byte trasferiti, che può essere, come per le normali funzioni di lettura e scrittura su file, inferiore a quelli richiesti; un valore negativo indicherà un errore mentre un valore nullo indicherà che non ci sono dati da trasferire (ad esempio si è giunti alla fine del file in lettura). Si tenga presente che, a seconda del verso del trasferimento dei dati, la funzione si comporta nei confronti del file descriptor che fa riferimento al file ordinario, come `read` o `write`, e pertanto potrà anche bloccarsi (a meno che non si sia aperto il suddetto file in modalità non bloccante).

I due argomenti `off_in` e `off_out` consentono di specificare, come per l'analogo `offset` di `sendfile`, la posizione all'interno del file da cui partire per il trasferimento dei dati. Come per `sendfile` un valore nullo indica di usare la posizione corrente sul file, ed essa sarà aggiornata automaticamente secondo il numero di byte trasferiti. Un valore non nullo invece deve essere un puntatore ad una variabile intera che indica la posizione da usare; questa verrà aggiornata, al ritorno della funzione, al byte successivo all'ultimo byte trasferito. Ovviamente soltanto uno di questi due argomenti, e più precisamente quello che fa riferimento al file descriptor non associato alla *pipe*, può essere specificato come valore non nullo.

⁵⁹si ricordi che questa funzione non è contemplata da nessuno standard, è presente solo su Linux, e pertanto deve essere evitata se si vogliono scrivere programmi portabili.

Infine l'argomento `flags` consente di controllare alcune caratteristiche del funzionamento della funzione; il contenuto è una maschera binaria e deve essere specificato come OR aritmetico dei valori riportati in tab. 10.19. Alcuni di questi valori vengono utilizzati anche dalle funzioni `vmsplice` e `tee` per cui la tabella riporta le descrizioni complete di tutti i valori possibili anche quando, come per `SPLICE_F_GIFT`, questi non hanno effetto su `splice`.

Valore	Significato
<code>SPLICE_F_MOVE</code>	Suggerisce al kernel di spostare le pagine di memoria contenenti i dati invece di copiarle: per una maggiore efficienza <code>splice</code> usa quando possibile i meccanismi della memoria virtuale per eseguire i trasferimenti di dati. In maniera analoga a <code>mmap</code>), qualora le pagine non possano essere spostate dalla <i>pipe</i> o il buffer non corrisponda a pagine intere esse saranno comunque copiate. Viene usato soltanto da <code>splice</code> .
<code>SPLICE_F_NONBLOCK</code>	Richiede di operare in modalità non bloccante; questo flag influisce solo sulle operazioni che riguardano l'I/O da e verso la <i>pipe</i> . Nel caso di <code>splice</code> questo significa che la funzione potrà comunque bloccarsi nell'accesso agli altri file descriptor (a meno che anch'essi non siano stati aperti in modalità non bloccante).
<code>SPLICE_F_MORE</code>	Indica al kernel che ci sarà l'invio di ulteriori dati in una <code>splice</code> successiva, questo è un suggerimento utile che viene usato quando <code>fd_out</code> è un socket. Questa opzione consente di utilizzare delle opzioni di gestione dei socket che permettono di ottimizzare le trasmissioni via rete (si veda la descrizione di <code>TCP_CORK</code> in sez. 16.2.5 e quella di <code>MSG_MORE</code> in sez. 18.1.2). Attualmente viene usato solo da <code>splice</code> , potrà essere implementato in futuro anche per <code>vmsplice</code> e <code>tee</code> .
<code>SPLICE_F_GIFT</code>	Le pagine di memoria utente sono “donate” al kernel; questo significa che la cache delle pagine e i dati su disco potranno differire, e che l'applicazione non potrà modificare quest'area di memoria. Se impostato una seguente <code>splice</code> che usa <code>SPLICE_F_MOVE</code> potrà spostare le pagine con successo, altrimenti esse dovranno essere copiate; per usare questa opzione i dati dovranno essere opportunamente allineati in posizione ed in dimensione alle pagine di memoria. Viene usato soltanto da <code>vmsplice</code> .

Tabella 10.19: Le costanti che identificano i bit della maschera binaria dell'argomento `flags` di `splice`, `vmsplice` e `tee`.

Per capire meglio il funzionamento di `splice` vediamo un esempio con un semplice programma che usa questa funzione per effettuare la copia di un file su un altro senza utilizzare buffer in *user space*. Lo scopo del programma è quello di eseguire la copia dei dati con `splice`, questo significa che si dovrà usare la funzione due volte, prima per leggere i dati dal file di ingresso e poi per scriverli su quello di uscita, appoggiandosi ad una *pipe*: lo schema del flusso dei dati è illustrato in fig. 10.19.

Figura 10.19: Struttura del flusso di dati usato dal programma `splice.c`.

Il programma si chiama `splice.c` ed il codice completo è disponibile coi sorgenti allegati alla guida, il corpo principale del programma, che non contiene la sezione di gestione delle opzioni, le funzioni di ausilio, le aperture dei file di ingresso e di uscita passati come argomenti e quella della *pipe* intermedia, è riportato in fig. 10.20.

Il ciclo principale (13-38) inizia con la lettura dal file sorgente tramite la prima `splice` (14-15), in questo caso si è usato come primo argomento il file descriptor del file sorgente e come terzo quello del capo in scrittura della *pipe*. Il funzionamento delle *pipe* e l'uso della coppia di file descriptor ad esse associati è trattato in dettaglio in sez. 11.1; non ne parleremo qui dato che nell'ottica dell'uso di `splice` questa operazione corrisponde semplicemente al trasferimento dei dati dal file al buffer in *kernel space*.

```

1 #define _GNU_SOURCE
2 #include <fcntl.h>          /* file control functions */
3 ...
4
5 int main(int argc, char *argv[])
6 {
7     int size = 4096;
8     int pipefd[2];
9     int in_fd, out_fd;
10    int nread, nwrite;
11    ...
12    /* copy loop */
13    while (1) {
14        nread = splice(in_fd, NULL, pipefd[1], NULL, size,
15                      SPLICE_F_MOVE|SPLICE_F_MORE);
16        if (nread == 0) break;
17        if (nread < 0) {
18            if (errno == EINTR) {
19                continue;
20            } else {
21                perror("read_error");
22                exit(EXIT_FAILURE);
23            }
24        }
25        while (nread > 0) {
26            nwrite = splice(pipefd[0], NULL, out_fd, NULL, nread,
27                           SPLICE_F_MOVE|SPLICE_F_MORE);
28            if (nwrite < 0) {
29                if (errno == EINTR)
30                    continue;
31                else {
32                    perror("write_error");
33                    exit(EXIT_FAILURE);
34                }
35            }
36            nread -= nwrite;
37        }
38    }
39    return EXIT_SUCCESS;
40 }

```

Figura 10.20: Esempio di codice che usa `splice` per effettuare la copia di un file.

La lettura viene eseguita in blocchi pari alla dimensione specificata dall'opzione `-s` (il default è 4096); essendo in questo caso `splice` equivalente ad una `read` sul file, se ne controlla il valore di uscita in `nread` che indica quanti byte sono stati letti, se detto valore è nullo (16) questo significa che si è giunti alla fine del file sorgente e pertanto l'operazione di copia è conclusa e si può uscire dal ciclo arrivando alla conclusione del programma (59). In caso di valore negativo (17-24) c'è stato un errore ed allora si ripete la lettura (16) se questo è dovuto ad una interruzione, o altrimenti si esce con un messaggio di errore (21-23).

Una volta completata con successo la lettura si avvia il ciclo di scrittura (25-37); questo inizia (26-27) con la seconda `splice` che cerca di scrivere gli `nread` byte letti, si noti come in questo caso il primo argomento faccia di nuovo riferimento alla *pipe* (in questo caso si usa il capo in lettura, per i dettagli si veda al solito sez. 11.1) mentre il terzo sia il file descriptor del file di destinazione.

Di nuovo si controlla il numero di byte effettivamente scritti restituito in `nwrite` e in caso di errore al solito si ripete la scrittura se questo è dovuto a una interruzione o si esce con un messaggio negli altri casi (28-35). Infine si chiude il ciclo di scrittura sottraendo (37) il numero di byte scritti a quelli di cui è richiesta la scrittura,⁶⁰ così che il ciclo di scrittura venga ripetuto fintanto che il valore risultante sia maggiore di zero, indice che la chiamata a `splice` non ha esaurito tutti i dati presenti sul buffer.

Si noti come il programma sia concettualmente identico a quello che si sarebbe scritto usando `read` al posto della prima `splice` e `write` al posto della seconda, utilizzando un buffer in *user space* per eseguire la copia dei dati, solo che in questo caso non è stato necessario allocare nessun buffer e non si è trasferito nessun dato in *user space*. Si noti anche come si sia usata la combinazione `SPLICE_F_MOVE | SPLICE_F_MORE` per l'argomento `flags` di `splice`, infatti anche se un valore nullo avrebbe dato gli stessi risultati, l'uso di questi flag, che si ricordi servono solo a dare suggerimenti al kernel, permette in genere di migliorare le prestazioni.

Come accennato con l'introduzione di `splice` sono state realizzate anche altre due *system call*, `vmsplice` e `tee`, che utilizzano la stessa infrastruttura e si basano sullo stesso concetto di manipolazione e trasferimento di dati attraverso un buffer in *kernel space*; benché queste non attengono strettamente ad operazioni di trasferimento dati fra file descriptor, le tratteremo qui, essendo strettamente correlate fra loro.

La prima funzione, `vmsplice`, è la più simile a `splice` e come indica il suo nome consente di trasferire i dati dalla memoria virtuale di un processo (ad esempio per un file mappato in memoria) verso una *pipe*; il suo prototipo è:

```
#include <fcntl.h>
#include <sys/uio.h>
long vmsplice(int fd, const struct iovec *iov, unsigned long nr_segs,
              unsigned int flags)
```

Trasferisce dati dalla memoria di un processo verso una *pipe*.

La funzione ritorna il numero di byte trasferiti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	o <code>fd</code> non è un file descriptor valido o non fa riferimento ad una <i>pipe</i> .
<code>EINVAL</code>	si è usato un valore nullo per <code>nr_segs</code> oppure si è usato <code>SPLICE_F_GIFT</code> ma la memoria non è allineata.
<code>ENOMEM</code>	non c'è memoria sufficiente per l'operazione richiesta.

La *pipe* indicata da `fd` dovrà essere specificata tramite il file descriptor corrispondente al suo capo aperto in scrittura (di nuovo si faccia riferimento a sez. 11.1), mentre per indicare quali segmenti della memoria del processo devono essere trasferiti verso di essa si dovrà utilizzare un vettore di strutture `iovec` (vedi fig. 10.18), esattamente con gli stessi criteri con cui le si usano per l'I/O vettorizzato, indicando gli indirizzi e le dimensioni di ciascun segmento di memoria su cui si vuole operare; le dimensioni del suddetto vettore devono essere passate nell'argomento `nr_segs` che indica il numero di segmenti di memoria da trasferire. Sia per il vettore che per il valore massimo di `nr_segs` valgono le stesse limitazioni illustrate in sez. 10.4.2.

In caso di successo la funzione ritorna il numero di byte trasferiti sulla *pipe*. In generale, se i dati una volta creati non devono essere riutilizzati (se cioè l'applicazione che chiama `vmsplice` non modificherà più la memoria trasferita), è opportuno utilizzare per `flag` il valore `SPLICE_F_GIFT`; questo fa sì che il kernel possa rimuovere le relative pagine dalla cache della memoria virtuale, così che queste possono essere utilizzate immediatamente senza necessità di eseguire una copia dei dati che contengono.

⁶⁰in questa parte del ciclo `nread`, il cui valore iniziale è dato dai byte letti dalla precedente chiamata a `splice`, viene ad assumere il significato di byte da scrivere.

La seconda funzione aggiunta insieme a `splice` è `tee`, che deve il suo nome all'omonimo comando in *user space*, perché in analogia con questo permette di duplicare i dati in ingresso su una *pipe* su un'altra *pipe*. In sostanza, sempre nell'ottica della manipolazione dei dati su dei buffer in *kernel space*, la funzione consente di eseguire una copia del contenuto del buffer stesso. Il prototipo di `tee` è il seguente:

```
#include <fcntl.h>
long tee(int fd_in, int fd_out, size_t len, unsigned int flags)
                                                Duplica i dati da una pipe ad un'altra.
```

La funzione ritorna restituisce il numero di byte copiati in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` o uno fra `fd_in` e `fd_out` non fa riferimento ad una *pipe* o entrambi fanno riferimento alla stessa *pipe*.

`ENOMEM` non c'è memoria sufficiente per l'operazione richiesta.

La funzione copia `len` byte del contenuto di una *pipe* su di un'altra; `fd_in` deve essere il capo in lettura della *pipe* sorgente e `fd_out` il capo in scrittura della *pipe* destinazione; a differenza di quanto avviene con `read` i dati letti con `tee` da `fd_in` non vengono consumati e restano disponibili sulla *pipe* per una successiva lettura (di nuovo per il comportamento delle *pipe* si veda sez. 11.1). Al momento⁶¹ il solo valore utilizzabile per `flag`, fra quelli elencati in tab. 10.19, è `SPLICE_F_NONBLOCK` che rende la funzione non bloccante.

La funzione restituisce il numero di byte copiati da una *pipe* all'altra (o `-1` in caso di errore), un valore nullo indica che non ci sono byte disponibili da copiare e che il capo in scrittura della *pipe* è stato chiuso; si tenga presente però che questo non avviene se si è impostato il flag `SPLICE_F_NONBLOCK`, in tal caso infatti si avrebbe un errore di `EAGAIN`. Un esempio di realizzazione del comando `tee` usando questa funzione, ripreso da quello fornito nella pagina di manuale e dall'esempio allegato al patch originale, è riportato in fig. 10.21. Il programma consente di copiare il contenuto dello *standard input* sullo *standard output* e su un file specificato come argomento, il codice completo si trova nel file `tee.c` dei sorgenti allegati alla guida.

La prima parte del programma, che si è omessa per brevità, si cura semplicemente di controllare che sia stato fornito almeno un argomento (il nome del file su cui scrivere), di aprirlo e che sia lo *standard input* che lo *standard output* corrispondano ad una *pipe*.

Il ciclo principale (11-32) inizia con la chiamata a `tee` che duplica il contenuto dello *standard input* sullo *standard output* (13), questa parte è del tutto analoga ad una lettura ed infatti come nell'esempio di fig. 10.20 si controlla il valore di ritorno della funzione in `len`; se questo è nullo significa che non ci sono più dati da leggere e si chiude il ciclo (14), se è negativo c'è stato un errore, ed allora si ripete la chiamata se questo è dovuto ad una interruzione (15-48) o si stampa un messaggio di errore e si esce negli altri casi (18-21).

Una volta completata la copia dei dati sullo *standard output* si possono estrarre dallo *standard input* e scrivere sul file, di nuovo su usa un ciclo di scrittura (24-31) in cui si ripete una chiamata a `splice` (25) fintanto che non si sono scritti tutti i `len` byte copiati in precedenza con `tee` (il funzionamento è identico all'analogo ciclo di scrittura del precedente esempio di fig. 10.20).

Infine una nota finale riguardo `splice`, `vmsplice` e `tee`: occorre sottolineare che benché finora si sia parlato di trasferimenti o copie di dati in realtà nella implementazione di queste *system call* non è affatto detto che i dati vengono effettivamente spostati o copiati, il kernel

⁶¹quello della stesura di questo paragrafo, avvenuta il Gennaio 2010, in futuro potrebbe essere implementato anche `SPLICE_F_MORE`.

```

1 #define _GNU_SOURCE
2 #include <fcntl.h>          /* file control functions */
3 ...
4 int main(int argc, char *argv[])
5 {
6     size_t size = 4096;
7     int fd, len, nwrite;
8     struct stat fdata;
9     ...
10    /* tee loop */
11    while (1) {
12        /* copy stdin to stdout */
13        len = tee(STDIN_FILENO, STDOUT_FILENO, size, 0);
14        if (len == 0) break;
15        if (len < 0) {
16            if (errno == EAGAIN) {
17                continue;
18            } else {
19                perror("error_on_tee_stdin_to_stdout");
20                exit(EXIT_FAILURE);
21            }
22        }
23        /* write data to the file using splice */
24        while (len > 0) {
25            nwrite = splice(STDIN_FILENO, NULL, fd, NULL, len, SPLICE_F_MOVE);
26            if (nwrite < 0) {
27                perror("error_on_splice_stdin_to_file");
28                break;
29            }
30            len -= nwrite;
31        }
32    }
33    exit(EXIT_SUCCESS);
34 }

```

Figura 10.21: Esempio di codice che usa tee per copiare i dati dello standard input sullo standard output e su un file.

infatti realizza le *pipe* come un insieme di puntatori⁶² alle pagine di memoria interna che contengono i dati, per questo una volta che i dati sono presenti nella memoria del kernel tutto quello che viene fatto è creare i suddetti puntatori ed aumentare il numero di referenze; questo significa che anche con tee non viene mai copiato nessun byte, vengono semplicemente copiati i puntatori.

10.4.4 Gestione avanzata dell'accesso ai dati dei file

Nell'uso generico dell'interfaccia per l'accesso al contenuto dei file le operazioni di lettura e scrittura non necessitano di nessun intervento di supervisione da parte dei programmi, si eseguirà una `read` o una `write`, i dati verranno passati al kernel che provvederà ad effettuare tutte le operazioni (e a gestire il *caching* dei dati) per portarle a termine in quello che ritiene essere il modo più efficiente.

⁶²per essere precisi si tratta di un semplice buffer circolare, un buon articolo sul tema si trova su <http://lwn.net/Articles/118750/>.

Il problema è che il concetto di migliore efficienza impiegato dal kernel è relativo all'uso generico, mentre esistono molti casi in cui ci sono esigenze specifiche dei singoli programmi, che avendo una conoscenza diretta di come verranno usati i file, possono necessitare di effettuare delle ottimizzazioni specifiche, relative alle proprie modalità di I/O sugli stessi. Tratteremo in questa sezione una serie di funzioni che consentono ai programmi di ottimizzare il loro accesso ai dati dei file e controllare la gestione del relativo *caching*.

Una prima funzione che può essere utilizzata per modificare la gestione ordinaria dell'I/O su un file è `readahead` (questa è una funzione specifica di Linux, introdotta con il kernel 2.4.13, e non deve essere usata se si vogliono scrivere programmi portabili), che consente di richiedere una lettura anticipata del contenuto dello stesso in cache, così che le seguenti operazioni di lettura non debbano subire il ritardo dovuto all'accesso al disco; il suo prototipo è:

```
#include <fcntl.h>
ssize_t readahead(int fd, off64_t *offset, size_t count)
    Esegue una lettura preventiva del contenuto di un file in cache.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF l'argomento `fd` non è un file descriptor valido o non è aperto in lettura.
EINVAL l'argomento `fd` si riferisce ad un tipo di file che non supporta l'operazione (come una *pipe* o un socket).

La funzione richiede che venga letto in anticipo il contenuto del file `fd` a partire dalla posizione `offset` e per un ammontare di `count` byte, in modo da portarlo in cache. La funzione usa la memoria virtuale ed il meccanismo della paginazione per cui la lettura viene eseguita in blocchi corrispondenti alle dimensioni delle pagine di memoria, ed i valori di `offset` e `count` vengono arrotondati di conseguenza.

La funzione estende quello che è un comportamento normale del kernel che, quando si legge un file, aspettandosi che l'accesso prosegua, esegue sempre una lettura preventiva di una certa quantità di dati; questo meccanismo di lettura anticipata viene chiamato *read-ahead*, da cui deriva il nome della funzione. La funzione `readahead`, per ottimizzare gli accessi a disco, effettua la lettura in cache della sezione richiesta e si blocca fintanto che questa non viene completata. La posizione corrente sul file non viene modificata ed indipendentemente da quanto indicato con `count` la lettura dei dati si interrompe una volta raggiunta la fine del file.

Si può utilizzare questa funzione per velocizzare le operazioni di lettura all'interno di un programma tutte le volte che si conosce in anticipo quanti dati saranno necessari nelle elaborazioni successive. Si potrà così concentrare in un unico momento (ad esempio in fase di inizializzazione) la lettura dei dati da disco, così da ottenere una migliore velocità di risposta nelle operazioni successive.

Il concetto di `readahead` viene generalizzato nello standard POSIX.1-2001 dalla funzione `posix_fadvise` (anche se l'argomento `len` è stato modificato da `size_t` a `off_t` nella revisione POSIX.1-2003 TC1) che consente di "avvisare" il kernel sulle modalità con cui si intende accedere nel futuro ad una certa porzione di un file, così che esso possa provvedere le opportune ottimizzazioni; il prototipo di `posix_fadvise`⁶³ è:

```
#include <fcntl.h>
int posix_fadvise(int fd, off_t offset, off_t len, int advice)
    Dichiarare al kernel le future modalità di accesso ad un file.
```

⁶³la funzione è stata introdotta su Linux solo a partire dal kernel 2.5.60, ed è disponibile soltanto se è stata definita la macro `_XOPEN_SOURCE` ad valore di almeno 600 o la macro `_POSIX_C_SOURCE` ad valore di almeno 200112L.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF	l'argomento <code>fd</code> non è un file descriptor valido.
EINVAL	il valore di <code>advice</code> non è valido o <code>fd</code> si riferisce ad un tipo di file che non supporta l'operazione (come una <i>pipe</i> o un socket).
ESPIPE	previsto dallo standard se <code>fd</code> è una <i>pipe</i> o un socket (ma su Linux viene restituito <code>EINVAL</code>).

La funzione dichiara al kernel le modalità con cui intende accedere alla regione del file indicato da `fd` che inizia alla posizione `offset` e si estende per `len` byte. Se per `len` si usa un valore nullo la regione coperta sarà da `offset` alla fine del file, ma questo è vero solo per le versioni più recenti, fino al kernel 2.6.6 il valore nullo veniva interpretato letteralmente. Le modalità sono indicate dall'argomento `advice` che è una maschera binaria dei valori illustrati in tab. 10.20, che riprendono il significato degli analoghi già visti in sez. 10.4.1 per `madvise`.⁶⁴ Si tenga presente comunque che la funzione dà soltanto un avvertimento, non esiste nessun vincolo per il kernel, che utilizza semplicemente l'informazione.

Valore	Significato
POSIX_FADV_NORMAL	Non ci sono avvisi specifici da fare riguardo le modalità di accesso, il comportamento sarà identico a quello che si avrebbe senza nessun avviso.
POSIX_FADV_SEQUENTIAL	L'applicazione si aspetta di accedere di accedere ai dati specificati in maniera sequenziale, a partire dalle posizioni più basse.
POSIX_FADV_RANDOM	I dati saranno letti in maniera completamente casuale.
POSIX_FADV_NOREUSE	I dati saranno acceduti una sola volta.
POSIX_FADV_WILLNEED	I dati saranno acceduti a breve.
POSIX_FADV_DONTNEED	I dati non saranno acceduti a breve.

Tabella 10.20: Valori delle costanti usabili per l'argomento `advice` di `posix_fadvise`, che indicano la modalità con cui si intende accedere ad un file.

Come `madvise` anche `posix_fadvise` si appoggia al sistema della memoria virtuale ed al meccanismo standard del *read-ahead* utilizzato dal kernel; in particolare utilizzando il valore `POSIX_FADV_SEQUENTIAL` si raddoppia la dimensione dell'ammontare di dati letti preventivamente rispetto al default, aspettandosi appunto una lettura sequenziale che li utilizzerà, mentre con `POSIX_FADV_RANDOM` si disabilita del tutto il suddetto meccanismo, dato che con un accesso del tutto casuale è inutile mettersi a leggere i dati immediatamente successivi gli attuali; infine l'uso di `POSIX_FADV_NORMAL` consente di riportarsi al comportamento di default.

Le due modalità `POSIX_FADV_NOREUSE` e `POSIX_FADV_WILLNEED` fino al kernel 2.6.18 erano equivalenti, a partire da questo kernel la prima viene non ha più alcun effetto, mentre la seconda dà inizio ad una lettura in cache della regione del file indicata. La quantità di dati che verranno letti è ovviamente limitata in base al carico che si viene a creare sul sistema della memoria virtuale, ma in genere una lettura di qualche megabyte viene sempre soddisfatta (ed un valore superiore è solo raramente di qualche utilità). In particolare l'uso di `POSIX_FADV_WILLNEED` si può considerare l'equivalente POSIX di *readahead*.

Infine con `POSIX_FADV_DONTNEED` si dice al kernel di liberare le pagine di cache occupate dai dati presenti nella regione di file indicata. Questa è una indicazione utile che permette di alleggerire il carico sulla cache, ed un programma può utilizzare periodicamente questa funzione per liberare pagine di memoria da dati che non sono più utilizzati per far posto a nuovi

⁶⁴dato che si tratta dello stesso tipo di funzionalità, in questo caso applicata direttamente al sistema ai contenuti di un file invece che alla sua mappatura in memoria.

dati utili; la pagina di manuale riporta l'esempio dello streaming di file di grosse dimensioni, dove le pagine occupate dai dati già inviati possono essere tranquillamente scartate.

Sia `posix_fadvise` che `readahead` attengono alla ottimizzazione dell'accesso in lettura; lo standard POSIX.1-2001 prevede anche una funzione specifica per le operazioni di scrittura, `posix_fallocate` (la funzione è stata introdotta a partire dalle glibc 2.1.94), che consente di preallocare dello spazio disco per assicurarsi che una seguente scrittura non fallisca, il suo prototipo, anch'esso disponibile solo se si definisce la macro `_XOPEN_SOURCE` ad almeno 600, è:

```
#include <fcntl.h>
int posix_fallocate(int fd, off_t offset, off_t len)
                                Richiede la allocazione di spazio disco per un file.
```

La funzione ritorna 0 in caso di successo e direttamente un codice di errore altrimenti, in tal caso `errno` non viene impostato, e si otterrà direttamente uno dei valori:

EBADF	l'argomento <code>fd</code> non è un file descriptor valido o non è aperto in scrittura.
EINVAL	<code>offset</code> o <code>len</code> sono minori di zero.
EFBIG	il valore di <code>(offset + len)</code> eccede la dimensione massima consentita per un file.
ENODEV	l'argomento <code>fd</code> non fa riferimento ad un file regolare.
ENOSPC	non c'è sufficiente spazio disco per eseguire l'operazione.
ESPIPE	l'argomento <code>fd</code> è una <i>pipe</i> .

La funzione assicura che venga allocato sufficiente spazio disco perché sia possibile scrivere sul file indicato dall'argomento `fd` nella regione che inizia dalla posizione `offset` e si estende per `len` byte; se questa regione si estende oltre la fine del file le dimensioni di quest'ultimo saranno incrementate di conseguenza. Dopo aver eseguito con successo la funzione è garantito che una successiva scrittura nella regione indicata non fallirà per mancanza di spazio disco. La funzione non ha nessun effetto né sul contenuto, né sulla posizione corrente del file.

Ci si può chiedere a cosa possa servire una funzione come `posix_fallocate` dato che è sempre possibile ottenere l'effetto voluto eseguendo esplicitamente sul file la scrittura di una serie di zeri (usando `pwrite` per evitare spostamenti della posizione corrente sul file) per l'estensione di spazio necessaria qualora il file debba essere esteso o abbia dei buchi.⁶⁵ In realtà questa è la modalità con cui la funzione veniva realizzata nella prima versione fornita dalla *glibc*, per cui la funzione costituiva in sostanza soltanto una standardizzazione delle modalità di esecuzione di questo tipo di allocazioni.

Questo metodo, anche se funzionante, comporta però l'effettiva esecuzione una scrittura su tutto lo spazio disco necessario, da fare al momento della richiesta di allocazione, pagandone il conseguente prezzo in termini di prestazioni; il tutto quando in realtà servirebbe solo poter riservare lo spazio per poi andarci a scrivere, una sola volta, quando il contenuto finale diventa effettivamente disponibile. Per poter fare tutto questo è però necessario il supporto da parte del kernel, e questo è divenuto disponibile solo a partire dal kernel 2.6.23 in cui è stata introdotta la nuova *system call* `fallocate`,⁶⁶ che consente di realizzare direttamente all'interno del kernel l'allocazione dello spazio disco così da poter realizzare una versione di `posix_fallocate` con prestazioni molto più elevate; nella *glibc* la nuova *system call* viene sfruttata per la realizzazione di `posix_fallocate` a partire dalla versione 2.10.

Trattandosi di una funzione di servizio, ed ovviamente disponibile esclusivamente su Linux,

⁶⁵si ricordi che occorre scrivere per avere l'allocazione e che l'uso di `truncate` per estendere un file creerebbe soltanto uno *sparse file* (vedi sez. 5.1.3) senza una effettiva allocazione dello spazio disco.

⁶⁶non è detto che la funzione sia disponibile per tutti i filesystem, ad esempio per XFS il supporto è stato introdotto solo a partire dal kernel 2.6.25.

inizialmente `fallocate` non era stata definita come funzione di libreria,⁶⁷ ma a partire dalla `glibc` 2.10 è stato fornito un supporto esplicito; il suo prototipo è:

```
#include <fcntl.h>
int fallocate(int fd, int mode, off_t offset, off_t len)
    Prealloca dello spazio disco per un file.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF `fd` non fa riferimento ad un file descriptor valido aperto in scrittura.

EFBIG la somma di `offset` e `len` eccede le dimensioni massime di un file.

EINVAL `offset` è minore di zero o `len` è minore o uguale a zero.

ENODEV `fd` non fa riferimento ad un file ordinario o a una directory.

EPERM il file è immutabile o *append-only* (vedi sez. 4.4.1).

ENOSYS il filesystem contenente il file associato a `fd` non supporta `fallocate`.

EOPNOTSUPP il filesystem contenente il file associato a `fd` non supporta l'operazione `mode`.

ed inoltre EINTR, EIO e ENOSPC nel loro significato generico.

La funzione prende gli stessi argomenti di `posix_fallocate` con lo stesso significato, a cui si aggiunge l'argomento `mode` che indica le modalità di allocazione; se questo è nullo il comportamento è identico a quello di `posix_fallocate` e si può considerare `fallocate` come l'implementazione ottimale della stessa a livello di kernel.

Inizialmente l'unico altro valore possibile per `mode` era `FALLOC_FL_KEEP_SIZE` che richiede che la dimensione del file (quella ottenuta nel campo `st_size` di una struttura `stat` dopo una chiamata a `fstat`) non venga modificata anche quando la somma di `offset` e `len` eccede la dimensione corrente, che serve quando si deve comunque preallocare dello spazio per scritture in append. In seguito sono stati introdotti altri valori, riassunti in tab.10.21, per compiere altre operazioni relative alla allocazione dello spazio disco dei file.

Valore	Significato
FALLOC_FL_INSERT	.
FALLOC_FL_COLLAPSE_RANGE	.
FALLOC_FL_KEEP_SIZE	Mantiene invariata la dimensione del file, pur allocando lo spazio disco anche oltre la dimensione corrente del file.
FALLOC_FL_PUNCH_HOLE	Crea un <i>buc</i> nel file (vedi sez. 5.1.3) rendendolo una <i>sparse file</i> (dal kernel 2.6.38).
FALLOC_FL_ZERO_RANGE	.

Tabella 10.21: Valori delle costanti usabili per l'argomento `mode` di `fallocate`.

In particolare con `FALLOC_FL_PUNCH_HOLE` è possibile scartare il contenuto della sezione di file indicata da `offset` e `len`, creando un *buc* (si ricordi quanto detto in sez. 5.1.3); i blocchi del file interamente contenuti nell'intervallo verranno disallocati, la parte di intervallo contenuta parzialmente in altri blocchi verrà riempita con zeri e la lettura dal file restituirà degli zeri per tutto l'intervallo indicato. In sostanza si rende il file uno *sparse file* a posteriori.

10.4.5 Altre funzionalità avanzate

da fare

⁶⁷pertanto poteva essere invocata soltanto in maniera indiretta con l'ausilio di `syscall`, vedi sez. 2.1.2, come `long fallocate(int fd, int mode, loff_t offset, loff_t len)`.

Capitolo 11

L'intercomunicazione fra processi

Uno degli aspetti fondamentali della programmazione in un sistema unix-like è la comunicazione fra processi. In questo capitolo affronteremo solo i meccanismi più elementari che permettono di mettere in comunicazione processi diversi, come quelli tradizionali che coinvolgono *pipe* e *fifo* e i meccanismi di intercomunicazione di System V e quelli POSIX.

Tralascieremo invece tutte le problematiche relative alla comunicazione attraverso la rete (e le relative interfacce) che saranno affrontate in dettaglio in un secondo tempo. Non affronteremo neanche meccanismi più complessi ed evoluti come le RPC (*Remote Procedure Calls*) che in genere sono implementati da un ulteriore livello di librerie sopra i meccanismi elementari.

11.1 L'intercomunicazione fra processi tradizionale

Il primo meccanismo di comunicazione fra processi introdotto nei sistemi Unix, è quello delle cosiddette *pipe*; esse costituiscono una delle caratteristiche peculiari del sistema, in particolare modo dell'interfaccia a linea di comando. In questa sezione descriveremo le sue basi, le funzioni che ne gestiscono l'uso e le varie forme in cui si è evoluto.

11.1.1 Le *pipe* standard

Le *pipe* nascono sostanzialmente con Unix, e sono il primo, e tuttora uno dei più usati, meccanismi di comunicazione fra processi. Si tratta in sostanza di una coppia di file descriptor connessi fra di loro in modo che quanto scrive su di uno si può rileggere dall'altro. Si viene così a costituire un canale di comunicazione realizzato tramite i due file descriptor, che costituisce appunto una sorta di *tubo* (che appunto il significato del termine inglese *pipe*) attraverso cui si possono far passare i dati.

In pratica si tratta di un buffer circolare in memoria in cui il kernel appoggia i dati immessi nel file descriptor su cui si scrive per farli poi riemergere dal file descriptor da cui si legge. Si tenga ben presente che in questo passaggio di dati non è previsto nessun tipo di accesso al disco e che nonostante l'uso dei file descriptor le *pipe* non hanno nulla a che fare con i file di dati di cui si è parlato al cap. 5.

La funzione di sistema che permette di creare questa speciale coppia di file descriptor associati ad una *pipe* è appunto *pipe*, ed il suo prototipo è:

```
#include <unistd.h>
int pipe(int filedes[2])
```

Crea la coppia di file descriptor di una *pipe*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EFAULT` `filedes` non è un indirizzo valido.

ed inoltre `EMFILE` e `ENFILE` nel loro significato generico.

La funzione restituisce la coppia di file descriptor nel vettore `filedes`, il primo è aperto in lettura ed il secondo in scrittura. Come accennato concetto di funzionamento di una *pipe* è semplice: quello che si scrive nel file descriptor aperto in scrittura viene ripresentato tale e quale nel file descriptor aperto in lettura.

I file descriptor infatti non sono connessi a nessun file reale, ma, come accennato, ad un buffer nel kernel la cui dimensione è specificata dal parametro di sistema `PIPE_BUF`, (vedi sez. 6.1.2). Lo schema di funzionamento di una *pipe* è illustrato in fig. 11.1, in cui sono indicati i due capi della *pipe*, associati a ciascun file descriptor, con le frecce che indicano la direzione del flusso dei dati.

Figura 11.1: Schema della struttura di una *pipe*.

Della funzione di sistema esiste una seconda versione, `pipe2`, introdotta con il kernel 2.6.27 e la *glibc* 2.9 e specifica di Linux (utilizzabile solo definendo la macro `_GNU_SOURCE`), che consente di impostare atomicamente le caratteristiche dei file descriptor restituiti, il suo prototipo è:

```
#include <unistd.h>
#include <fcntl.h>
int pipe2(int pipefd[2], int flags)
```

Crea la coppia di file descriptor di una *pipe*.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` il valore di `flags` non valido.

e gli altri già visti per `pipe` con lo stesso significato.

Utilizzando un valore nullo per `flags` la funzione è identica a `pipe`, si può però passare come valore l'OR aritmetico di uno qualunque fra `O_NONBLOCK` o `O_CLOEXEC` che hanno l'effetto di impostare su entrambi i file descriptor restituiti dalla funzione i relativi flag, già descritti per `open` in tab. 5.4, che attivano rispettivamente la modalità di accesso *non-bloccante* ed il *close-on-exec*.

Chiaramente creare una *pipe* all'interno di un singolo processo non serve a niente; se però ricordiamo quanto esposto in sez. 5.2.1 riguardo al comportamento dei file descriptor nei processi figli, è immediato capire come una *pipe* possa diventare un meccanismo di intercomunicazione. Un processo figlio infatti condivide gli stessi file descriptor del padre, compresi quelli associati ad una *pipe* (secondo la situazione illustrata in fig. 11.2). In questo modo se uno dei processi scrive su un capo della *pipe*, l'altro può leggere.

Figura 11.2: Schema dei collegamenti ad una *pipe*, condivisi fra processo padre e figlio dopo l'esecuzione `fork`.

Tutto ciò ci mostra come sia immediato realizzare un meccanismo di comunicazione fra processi attraverso una *pipe*, utilizzando le proprietà ordinarie dei file, ma ci mostra anche qual

è il principale limite nell'uso delle *pipe*.¹ È necessario infatti che i processi possano condividere i file descriptor della *pipe*, e per questo essi devono comunque essere *parenti* (dall'inglese *siblings*), cioè o derivare da uno stesso processo padre in cui è avvenuta la creazione della *pipe*, o, più comunemente, essere nella relazione padre/figlio.

A differenza di quanto avviene con i file normali, la lettura da una *pipe* può essere bloccante (qualora non siano presenti dati), inoltre se si legge da una *pipe* il cui capo in scrittura è stato chiuso, si avrà la ricezione di un EOF (vale a dire che la funzione `read` ritornerà restituendo 0). Se invece si esegue una scrittura su una *pipe* il cui capo in lettura non è aperto il processo riceverà il segnale SIGPIPE, e la funzione di scrittura restituirà un errore di EPIPE (al ritorno del gestore, o qualora il segnale sia ignorato o bloccato).

La dimensione del buffer della *pipe* (PIPE_BUF) ci dà inoltre un'altra importante informazione riguardo il comportamento delle operazioni di lettura e scrittura su di una *pipe*; esse infatti sono atomiche fintanto che la quantità di dati da scrivere non supera questa dimensione. Qualora ad esempio si effettui una scrittura di una quantità di dati superiore l'operazione verrà effettuata in più riprese, consentendo l'intromissione di scritture effettuate da altri processi.

La dimensione originale del buffer era di 4096 byte (uguale ad una pagina di memoria) fino al kernel 2.6.11, ed è stata portata in seguito a 64kb; ma a partire dal kernel 2.6.35 è stata resa disponibile l'operazione di controllo `F_SETPIPE_SZ` (vedi sez. 11.1.1) che consente di modificarne la dimensione.

11.1.2 Un esempio dell'uso delle *pipe*

Per capire meglio il funzionamento delle *pipe* faremo un esempio di quello che è il loro uso più comune, analogo a quello effettuato della shell, e che consiste nell'inviare l'output di un processo (lo standard output) sull'input di un altro. Realizzeremo il programma di esempio nella forma di un *CGI*² che genera una immagine JPEG di un codice a barre, specificato come argomento in ingresso.

Un programma che deve essere eseguito come *CGI* deve rispondere a delle caratteristiche specifiche, esso infatti non viene lanciato da una shell, ma dallo stesso web server, alla richiesta di una specifica URL, che di solito ha la forma:

`http://www.sito.it/cgi-bin/programma?argomento`

ed il risultato dell'elaborazione deve essere presentato (con una intestazione che ne descrive il *mime-type*) sullo *standard output*, in modo che il server web possa reinviarlo al browser che ha effettuato la richiesta, che in questo modo è in grado di visualizzarlo opportunamente.

Figura 11.3: Schema dell'uso di una *pipe* come mezzo di comunicazione fra due processi attraverso l'esecuzione una *fork* e la chiusura dei capi non utilizzati.

Per realizzare quanto voluto useremo in sequenza i programmi `barcode` e `gs`, il primo infatti è in grado di generare immagini PostScript di codici a barre corrispondenti ad una qualunque stringa, mentre il secondo serve per poter effettuare la conversione della stessa immagine in formato JPEG. Usando una *pipe* potremo inviare l'output del primo sull'input

¹Stevens in [?] riporta come limite anche il fatto che la comunicazione è unidirezionale, ma in realtà questo è un limite superabile usando una coppia di *pipe*, anche se al costo di una maggiore complessità di gestione.

²quella dei CGI (*Common Gateway Interface*) è una interfaccia che consente ad un server web di eseguire un programma il cui output (che deve essere opportunamente formattato seguendo le specifiche dell'interfaccia) può essere presentato come risposta ad una richiesta HTTP al posto del contenuto di un file, e che ha costituito probabilmente la prima modalità con cui sono state create pagine HTML dinamiche.

del secondo, secondo lo schema mostrato in fig. 11.3, in cui la direzione del flusso dei dati è data dalle frecce continue.

Si potrebbe obiettare che sarebbe molto più semplice salvare il risultato intermedio su un file temporaneo. Questo però non tiene conto del fatto che un *CGI* può essere eseguito più volte in contemporanea, e si avrebbe una evidente *race condition* in caso di accesso simultaneo a detto file da istanze diverse. Il problema potrebbe essere superato utilizzando un sempre diverso per il file temporaneo, che verrebbe creato all'avvio di ogni istanza, utilizzato dai sottoprocessi, e cancellato alla fine della sua esecuzione; ma a questo punto le cose non sarebbero più tanto semplici. L'uso di una *pipe* invece permette di risolvere il problema in maniera semplice ed elegante, oltre ad essere molto più efficiente, dato che non si deve scrivere su disco.

Il programma ci servirà anche come esempio dell'uso delle funzioni di duplicazione dei file descriptor che abbiamo trattato in sez. 5.2.2, in particolare di `dup2`. È attraverso queste funzioni infatti che è possibile dirottare gli stream standard dei processi (che abbiamo visto in tab. 5.1 e sez. 5.3.1) sulla *pipe*. In fig. 11.4 abbiamo riportato il corpo del programma, il cui codice completo è disponibile nel file `BarCodePage.c` che si trova nella directory dei sorgenti.

La prima operazione del programma (4-12) è quella di creare le due *pipe* che serviranno per la comunicazione fra i due comandi utilizzati per produrre il codice a barre; si ha cura di controllare la riuscita della chiamata, inviando in caso di errore un messaggio invece dell'immagine richiesta. La funzione `WriteMess` non è riportata in fig. 11.4; essa si incarica semplicemente di formattare l'uscita alla maniera dei CGI, aggiungendo l'opportuno *mime-type*, e formattando il messaggio in HTML, in modo che quest'ultimo possa essere visualizzato correttamente da un browser.

Una volta create le *pipe*, il programma può creare (13-17) il primo processo figlio, che si incaricherà (19-25) di eseguire `barcode`. Quest'ultimo legge dallo standard input una stringa di caratteri, la converte nell'immagine PostScript del codice a barre ad essa corrispondente, e poi scrive il risultato direttamente sullo standard output.

Per poter utilizzare queste caratteristiche prima di eseguire `barcode` si chiude (20) il capo aperto in scrittura della prima *pipe*, e se ne collega (21) il capo in lettura allo *standard input* usando `dup2`. Si ricordi che invocando `dup2` il secondo file, qualora risulti aperto, viene, come nel caso corrente, chiuso prima di effettuare la duplicazione. Allo stesso modo, dato che `barcode` scrive l'immagine PostScript del codice a barre sullo standard output, per poter effettuare una ulteriore redirectione il capo in lettura della seconda *pipe* viene chiuso (22) mentre il capo in scrittura viene collegato allo standard output (23).

In questo modo all'esecuzione (25) di `barcode` (cui si passa in `size` la dimensione della pagina per l'immagine) quest'ultimo leggerà dalla prima *pipe* la stringa da codificare che gli sarà inviata dal padre, e scriverà l'immagine PostScript del codice a barre sulla seconda.

Al contempo una volta lanciato il primo figlio, il processo padre prima chiude (26) il capo inutilizzato della prima *pipe* (quello in ingresso) e poi scrive (27) la stringa da convertire sul capo in uscita, così che `barcode` possa riceverla dallo *standard input*. A questo punto l'uso della prima *pipe* da parte del padre è finito ed essa può essere definitivamente chiusa (28), si attende poi (29) che l'esecuzione di `barcode` sia completata.

Alla conclusione della sua esecuzione `barcode` avrà inviato l'immagine PostScript del codice a barre sul capo in scrittura della seconda *pipe*; a questo punto si può eseguire la seconda conversione, da PS a JPEG, usando il programma `gs`. Per questo si crea (30-34) un secondo processo figlio, che poi (35-42) eseguirà questo programma leggendo l'immagine PostScript creata da `barcode` dallo *standard input*, per convertirla in JPEG.

Per fare tutto ciò anzitutto si chiude (37) il capo in scrittura della seconda *pipe*, e se ne collega (38) il capo in lettura allo *standard input*. Per poter formattare l'output del programma

```

1 int main(int argc, char *argv[], char *envp[])
2 {
3     ...
4     /* create two pipes, pipein and pipeout, to handle communication */
5     if ( (retval = pipe(pipein)) ) {
6         WriteMess("input_pipe_creation_error");
7         exit(0);
8     }
9     if ( (retval = pipe(pipeout)) ) {
10        WriteMess("output_pipe_creation_error");
11        exit(0);
12    }
13    /* First fork: use child to run barcode program */
14    if ( (pid = fork()) == -1) { /* on error exit */
15        WriteMess("child_creation_error");
16        exit(0);
17    }
18    /* if child */
19    if (pid == 0) {
20        close(pipein[1]); /* close pipe write end */
21        dup2(pipein[0], STDIN_FILENO); /* remap stdin to pipe read end */
22        close(pipeout[0]);
23        dup2(pipeout[1], STDOUT_FILENO); /* remap stdout in pipe output */
24        execlp("barcode", "barcode", size, NULL);
25    }
26    close(pipein[0]); /* close input side of input pipe */
27    write(pipein[1], argv[1], strlen(argv[1])); /* write parameter to pipe */
28    close(pipein[1]); /* closing write end */
29    waitpid(pid, NULL, 0); /* wait child completion */
30    /* Second fork: use child to run ghostscript */
31    if ( (pid = fork()) == -1) {
32        WriteMess("child_creation_error");
33        exit(0);
34    }
35    /* second child, convert PS to JPEG */
36    if (pid == 0) {
37        close(pipeout[1]); /* close write end */
38        dup2(pipeout[0], STDIN_FILENO); /* remap read end to stdin */
39        /* send mime type */
40        write(STDOUT_FILENO, content, strlen(content));
41        execlp("gs", "gs", "-q", "-sDEVICE=jpeg", "-sOutputFile=-", "-", NULL);
42    }
43    /* still parent */
44    close(pipeout[1]);
45    waitpid(pid, NULL, 0);
46    exit(0);
47 }

```

Figura 11.4: Sezione principale del codice del CGI BarCodePage.c.

in maniera utilizzabile da un browser, si provvede anche 40) alla scrittura dell'apposita stringa di identificazione del *mime-type* in testa allo *standard output*. A questo punto si può invocare 41) *gs*, provvedendo le opportune opzioni del comando che consentono di leggere il file da convertire dallo *standard input* e di inviare la conversione sullo *standard output*.

Per completare le operazioni il processo padre chiude (44) il capo in scrittura della seconda *pipe*, e attende la conclusione del figlio (45); a questo punto può (46) uscire. Si tenga conto

che l'operazione di chiudere il capo in scrittura della seconda *pipe* è necessaria, infatti, se non venisse chiusa, **gs**, che legge il suo *standard input* da detta *pipe*, resterebbe bloccato in attesa di ulteriori dati in ingresso (l'unico modo che un programma ha per sapere che i dati in ingresso sono terminati è rilevare che lo *standard input* è stato chiuso), e la **wait** non ritornerebbe.

11.1.3 Le funzioni **popen** e **pclose**

Come si è visto la modalità più comune di utilizzo di una *pipe* è quella di utilizzarla per fare da tramite fra output ed input di due programmi invocati in sequenza; per questo motivo lo standard POSIX.2 ha introdotto due funzioni che permettono di sintetizzare queste operazioni. La prima di esse si chiama **popen** ed il suo prototipo è:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type)
    Esegue un programma dirottando l'uscita su una pipe.
```

La funzione ritorna l'indirizzo dello stream associato alla *pipe* in caso di successo e NULL per un errore, nel qual caso **errno** potrà assumere i valori relativi alle sottostanti invocazioni di *pipe* e *fork* o **EINVAL** se *type* non è valido.

La funzione crea una *pipe*, esegue una **fork** creando un nuovo processo nel quale invoca il programma **command** attraverso la shell (in sostanza esegue **/bin/sh** con il flag **-c**). L'argomento **type** deve essere una delle due stringhe **"w"** o **"r"**, per richiedere che la *pipe* restituita come valore di ritorno sia collegata allo *standard input* o allo *standard output* del comando invocato.

La funzione restituisce il puntatore ad uno stream associato alla *pipe* creata, che sarà aperto in sola lettura (e quindi associato allo *standard output* del programma indicato) in caso si sia indicato **r**, o in sola scrittura (e quindi associato allo *standard input*) in caso di **w**. A partire dalla versione 2.9 della *glibc* (questa è una estensione specifica di Linux) all'argomento **type** può essere aggiunta la lettera **"e"** per impostare automaticamente il flag di *close-on-exec* sul file descriptor sottostante.

Lo *stream* restituito da **popen** è identico a tutti gli effetti ai *file stream* visti in sez. 5.3, anche se è collegato ad una *pipe* e non ad un file, e viene sempre aperto in modalità *fully-buffered* (vedi sez. 5.3.2); l'unica differenza con gli usuali stream è che dovrà essere chiuso dalla seconda delle due nuove funzioni, **pclose**, il cui prototipo è:

```
#include <stdio.h>
int pclose(FILE *stream)
    Chiude una pipe creata con popen.
```

La funzione ritorna lo stato del processo creato da **popen** in caso di successo e **-1** per un errore, nel qual caso **errno** assumerà i valori derivanti dalle sottostanti funzioni **fclose** e **wait4**.

La funzione chiude il file **stream** associato ad una *pipe* creato da una precedente **popen**, ed oltre alla chiusura dello stream si incarica anche di attendere (tramite **wait4**) la conclusione del processo creato dalla precedente **popen**. Se lo stato di uscita non può essere letto la funzione restituirà per **errno** un errore di **ECHILD**.

Per illustrare l'uso di queste due funzioni riprendiamo il problema precedente: il programma mostrato in fig. 11.4 per quanto funzionante, è volutamente codificato in maniera piuttosto complessa, inoltre doveva scontare un problema di **gs** che non era in grado di riconoscere correttamente l'Encapsulated PostScript,³ per cui si era utilizzato il PostScript semplice, ge-

³si fa riferimento alla versione di GNU Ghostscript 6.53 (2002-02-13), usata quando l'esempio venne scritto per la prima volta.

nerando una pagina intera invece che una immagine delle dimensioni corrispondenti al codice a barre.

Se si vuole generare una immagine di dimensioni appropriate si deve usare un approccio diverso. Una possibilità sarebbe quella di ricorrere ad ulteriore programma, `epstopdf`, per convertire in PDF un file EPS (che può essere generato da `barcode` utilizzando l'opzione `-E`). Utilizzando un PDF al posto di un EPS `gs` esegue la conversione rispettando le dimensioni originarie del codice a barre e produce un JPEG di dimensioni corrette.

Questo approccio però non può funzionare per via di una delle caratteristiche principali delle *pipe*. Per poter effettuare la conversione di un PDF infatti è necessario, per la struttura del formato, potersi spostare (con `lseek`) all'interno del file da convertire. Se si esegue la conversione con `gs` su un file regolare non ci sono problemi, una *pipe* però è rigidamente sequenziale, e l'uso di `lseek` su di essa fallisce sempre con un errore di `ESPIPE`, rendendo impossibile la conversione. Questo ci dice che in generale la concatenazione di vari programmi funzionerà soltanto quando tutti prevedono una lettura sequenziale del loro input.

Per questo motivo si è dovuto utilizzare un procedimento diverso, eseguendo prima la conversione (sempre con `gs`) del PS in un altro formato intermedio, il PPM,⁴ dal quale poi si può ottenere un'immagine di dimensioni corrette attraverso vari programmi di manipolazione (`pnmcrop`, `pnmmargin`) che può essere infine trasformata in PNG (con `pnm2png`).

In questo caso però occorre eseguire in sequenza ben quattro comandi diversi, inviando l'uscita di ciascuno all'ingresso del successivo, per poi ottenere il risultato finale sullo *standard output*: un caso classico di utilizzazione delle *pipe*, in cui l'uso di `popen` e `pclose` permette di semplificare notevolmente la stesura del codice.

Nel nostro caso, dato che ciascun processo deve scrivere la sua uscita sullo *standard input* del successivo, occorrerà usare `popen` aprendo la *pipe* in scrittura. Il codice del nuovo programma è riportato in fig. 11.5. Come si può notare l'ordine di invocazione dei programmi è l'inverso di quello in cui ci si aspetta che vengano effettivamente eseguiti. Questo non comporta nessun problema dato che la lettura su una *pipe* è bloccante, per cui un processo, anche se lanciato per primo, se non ottiene i dati che gli servono si bloccherà in attesa sullo *standard input* finché non otterrà il risultato dell'elaborazione del processo che li deve creare, che pur essendo logicamente precedente, viene lanciato dopo di lui.

Nel nostro caso il primo passo (14) è scrivere il *mime-type* sullo *standard output*; a questo punto il processo padre non necessita più di eseguire ulteriori operazioni sullo *standard output* e può tranquillamente provvedere alla redirectione.

Dato che i vari programmi devono essere lanciati in successione, si è approntato un ciclo (15-19) che esegue le operazioni in sequenza: prima crea una *pipe* (17) per la scrittura eseguendo il programma con `popen`, in modo che essa sia collegata allo *standard input*, e poi redirige (18) lo *standard output* su detta *pipe*.

In questo modo il primo processo ad essere invocato (che è l'ultimo della catena) scriverà ancora sullo *standard output* del processo padre, ma i successivi, a causa di questa redirectione, scriveranno sulla *pipe* associata allo *standard input* del processo invocato nel ciclo precedente.

Alla fine tutto quello che resta da fare è lanciare (21) il primo processo della catena, che nel caso è `barcode`, e scrivere (23) la stringa del codice a barre sulla *pipe*, che è collegata al suo *standard input*, infine si può eseguire (24-27) un ciclo che chiuda con `pclose`, nell'ordine inverso rispetto a quello in cui le si sono create, tutte le *pipe* create in precedenza.

⁴il *Portable Pixmap file format* è un formato usato spesso come formato intermedio per effettuare conversioni, è infatti molto facile da manipolare, dato che usa caratteri ASCII per memorizzare le immagini, anche se per questo è estremamente inefficiente come formato di archiviazione.

```

1 int main(int argc, char *argv[], char *envp[])
2 {
3     FILE *pipe[4];
4     FILE *pipein;
5     char *cmd_string[4]={
6         "pnmtopng",
7         "pnmmargin_-white_10",
8         "pnmcrop",
9         "gs_-sDEVICE=ppmraw_-sOutputFile=-_-sNOPAUSE_-q_-_-c_showpage_-c_quit"
10    };
11    char content[]="Content-type:image/png\n\n";
12    int i;
13    /* write mime-type to stdout */
14    write(STDOUT_FILENO, content, strlen(content));
15    /* execute chain of command */
16    for (i=0; i<4; i++) {
17        pipe[i] = popen(cmd_string[i], "w");
18        dup2(fileno(pipe[i]), STDOUT_FILENO);
19    }
20    /* create barcode (in PS) */
21    pipein = popen("barcode", "w");
22    /* send barcode string to barcode program */
23    write(fileno(pipein), argv[1], strlen(argv[1]));
24    /* close all pipes (in reverse order) */
25    for (i=4; i==0; i--) {
26        pclose((pipe[i]));
27    }
28    exit(0);
29 }

```

Figura 11.5: Codice completo del CGI BarCode.c.

11.1.4 Le *pipe* con nome, o *fifo*

Come accennato in sez. 11.1.1 il problema delle *pipe* è che esse possono essere utilizzate solo da processi con un progenitore comune o nella relazione padre/figlio. Per superare questo problema lo standard POSIX.1 ha introdotto le *fifo*, che hanno le stesse caratteristiche delle *pipe*, ma che invece di essere visibili solo attraverso un file descriptor creato all'interno di un processo da una *system call* apposita, costituiscono un oggetto che risiede sul filesystem (si rammenti quanto detto in sez. 1.2.3) che può essere aperto come un qualunque file, così che i processi le possono usare senza dovere per forza essere in una relazione di *parentela*.

Utilizzando una *fifo* tutti i dati passeranno, come per le *pipe*, attraverso un buffer nel kernel, senza transitare dal filesystem. Il fatto che siano associate ad un *inode* presente sul filesystem serve infatti solo a fornire un punto di accesso per i processi, che permetta a questi ultimi di accedere alla stessa *fifo* senza avere nessuna relazione, con una semplice *open*. Il comportamento delle funzioni di lettura e scrittura è identico a quello illustrato per le *pipe* in sez. 11.1.1.

Abbiamo già trattato in sez. 4.2.5 le funzioni *mknod* e *mkfifo* che permettono di creare una *fifo*. Per utilizzarne una un processo non avrà che da aprire il relativo file speciale o in lettura o scrittura; nel primo caso il processo sarà collegato al capo di uscita della *fifo*, e dovrà leggere, nel secondo al capo di ingresso, e dovrà scrivere.

Il kernel alloca un singolo buffer per ciascuna *fifo* che sia stata aperta, e questa potrà essere acceduta contemporaneamente da più processi, sia in lettura che in scrittura. Dato che

per funzionare deve essere aperta in entrambe le direzioni, per una *fifo* la funzione `open` di norma si blocca se viene eseguita quando l'altro capo non è aperto.

Le *fifo* però possono essere anche aperte in modalità *non-bloccante*, nel qual caso l'apertura del capo in lettura avrà successo solo quando anche l'altro capo è aperto, mentre l'apertura del capo in scrittura restituirà l'errore di `ENXIO` fintanto che non verrà aperto il capo in lettura.

In Linux è possibile aprire le *fifo* anche in lettura/scrittura (lo standard POSIX lascia indefinito il comportamento in questo caso) operazione che avrà sempre successo immediato qualunque sia la modalità di apertura, bloccante e non bloccante. Questo può essere utilizzato per aprire comunque una *fifo* in scrittura anche se non ci sono ancora processi in lettura. Infine è possibile anche usare la *fifo* all'interno di un solo processo, nel qual caso però occorre stare molto attenti alle possibili situazioni di stallo: se si cerca di leggere da una *fifo* che non contiene dati si avrà infatti un *deadlock* immediato, dato che il processo si blocca e quindi non potrà mai eseguire le funzioni di scrittura.

Per la loro caratteristica di essere accessibili attraverso il filesystem, è piuttosto frequente l'utilizzo di una *fifo* come canale di comunicazione nelle situazioni in cui un processo deve ricevere informazioni da altri. In questo caso è fondamentale che le operazioni di scrittura siano atomiche; per questo si deve sempre tenere presente che questo è vero soltanto fintanto che non si supera il limite delle dimensioni di `PIPE_BUF` (si ricordi quanto detto in sez. 11.1.1).

A parte il caso precedente, che resta probabilmente il più comune, Stevens riporta in [?] altre due casistiche principali per l'uso delle *fifo*:

- Da parte dei comandi di shell, per evitare la creazione di file temporanei quando si devono inviare i dati di uscita di un processo sull'input di parecchi altri (attraverso l'uso del comando `tee`).
- Come canale di comunicazione fra un client ed un server (il modello *client-server* è illustrato in sez. 13.1.1).

Nel primo caso quello che si fa è creare tante *fifo* da usare come *standard input* quanti sono i processi a cui si vogliono inviare i dati; questi ultimi saranno stati posti in esecuzione indirizzando lo *standard input* dalle *fifo*, si potrà poi eseguire il processo che fornisce l'output replicando quest'ultimo, con il comando `tee`, sulle varie *fifo*.

Il secondo caso è relativamente semplice qualora si debba comunicare con un processo alla volta, nel qual caso basta usare due *fifo*, una per leggere ed una per scrivere. Le cose diventano invece molto più complesse quando si vuole effettuare una comunicazione fra un server ed un numero imprecisato di client. Se il primo infatti può ricevere le richieste attraverso una *fifo* “nota”, per le risposte non si può fare altrettanto, dato che, per la struttura sequenziale delle *fifo*, i client dovrebbero sapere prima di leggerli quando i dati inviati sono destinati a loro.

Per risolvere questo problema, si può usare un'architettura come quella illustrata in fig. 11.6 in cui i client inviano le richieste al server su una *fifo* nota mentre le risposte vengono reinviolate dal server a ciascuno di essi su una *fifo* temporanea creata per l'occasione.

Figura 11.6: Schema dell'utilizzo delle *fifo* nella realizzazione di una architettura di comunicazione client/server.

Come esempio di uso questa architettura e dell'uso delle *fifo*, abbiamo scritto un server di *fortunes*, che restituisce, alle richieste di un client, un detto a caso estratto da un insieme di frasi. Sia il numero delle frasi dell'insieme, che i file da cui esse vengono lette all'avvio, sono impostabili da riga di comando. Il corpo principale del server è riportato in fig. 11.7, dove si è tralasciata la parte che tratta la gestione delle opzioni a riga di comando, che effettua l'impostazione delle variabili `fortune_filename`, che indica il file da cui leggere le frasi, ed `n`,

che indica il numero di frasi tenute in memoria, ad un valore diverso da quelli preimpostati. Il codice completo è nel file `FortuneServer.c`.

```

1 char *fifoname = "/tmp/fortune.fifo";
2 int main(int argc, char *argv[])
3 {
4     /* Variables definition */
5     int i, n = 0;
6     char *fortunefilename = "/usr/share/games/fortunes/linux";
7     char **fortune;
8     char line[80];
9     int fifo_server, fifo_client;
10    int nread;
11    ...
12    if (n==0) usage(); /* if no pool depth exit printing usage info */
13    Signal(SIGTERM, HandSIGTERM); /* set handlers for termination */
14    Signal(SIGINT, HandSIGTERM);
15    Signal(SIGQUIT, HandSIGTERM);
16    i = FortuneParse(fortunefilename, fortune, n); /* parse phrases */
17    if (mkfifo(fifoname, 0622)) { /* create well known fifo if doesn't exist */
18        if (errno!=EEXIST) {
19            perror("Cannot_create_well_known_fifo");
20            exit(1);
21        }
22    }
23    daemon(0, 0);
24    /* open fifo two times to avoid EOF */
25    fifo_server = open(fifoname, O_RDONLY);
26    if (fifo_server < 0) {
27        perror("Cannot_open_read_only_well_known_fifo");
28        exit(1);
29    }
30    if (open(fifoname, O_WRONLY) < 0) {
31        perror("Cannot_open_write_only_well_known_fifo");
32        exit(1);
33    }
34    /* Main body: loop over requests */
35    while (1) {
36        nread = read(fifo_server, line, 79); /* read request */
37        if (nread < 0) {
38            perror("Read_Error");
39            exit(1);
40        }
41        line[nread] = 0; /* terminate fifo name string */
42        n = random() % i; /* select random value */
43        fifo_client = open(line, O_WRONLY); /* open client fifo */
44        if (fifo_client < 0) {
45            perror("Cannot_open");
46            exit(1);
47        }
48        nread = write(fifo_client, /* write phrase */
49                    fortune[n], strlen(fortune[n])+1);
50        close(fifo_client); /* close client fifo */
51    }
52 }

```

Figura 11.7: Sezione principale del codice del server di *fortunes* basato sulle *fifo*.

Il server richiede (12) che sia stata impostata una dimensione dell'insieme delle frasi non nulla, dato che l'inizializzazione del vettore `fortune` avviene solo quando questa dimensione viene specificata, la presenza di un valore nullo provoca l'uscita dal programma attraverso la funzione (non riportata) che ne stampa le modalità d'uso. Dopo di che installa (13-15) la funzione che gestisce i segnali di interruzione (anche questa non è riportata in fig. 11.7) che si limita a rimuovere dal filesystem la *fifo* usata dal server per comunicare.

Terminata l'inizializzazione (16) si effettua la chiamata alla funzione `FortuneParse` che legge dal file specificato in `fortunefilename` le prime *n* frasi e le memorizza (allocando dinamicamente la memoria necessaria) nel vettore di puntatori `fortune`. Anche il codice della funzione non è riportato, in quanto non direttamente attinente allo scopo dell'esempio.

Il passo successivo (17-22) è quello di creare con `mkfifo` la *fifo* nota sulla quale il server ascolterà le richieste, qualora si riscontri un errore il server uscirà (escludendo ovviamente il caso in cui la funzione `mkfifo` fallisce per la precedente esistenza della *fifo*).

Una volta che si è certi che la *fifo* di ascolto esiste la procedura di inizializzazione è completata. A questo punto (23) si può chiamare la funzione `daemon` per far proseguire l'esecuzione del programma in background come demone. Si può quindi procedere (24-33) alla apertura della *fifo*: si noti che questo viene fatto due volte, prima in lettura e poi in scrittura, per evitare di dover gestire all'interno del ciclo principale il caso in cui il server è in ascolto ma non ci sono client che effettuano richieste. Si ricordi infatti che quando una *fifo* è aperta solo dal capo in lettura, l'esecuzione di `read` ritorna con zero byte (si ha cioè una condizione di *end-of-file*).

Nel nostro caso la prima apertura si bloccherà fintanto che un qualunque client non apre a sua volta la *fifo* nota in scrittura per effettuare la sua richiesta. Pertanto all'inizio non ci sono problemi, il client però, una volta ricevuta la risposta, uscirà, chiudendo tutti i file aperti, compresa la *fifo*. A questo punto il server resta (se non ci sono altri client che stanno effettuando richieste) con la *fifo* chiusa sul lato in lettura, ed in questo stato la funzione `read` non si bloccherà in attesa di dati in ingresso, ma ritornerà in continuazione, restituendo una condizione di *end-of-file*.

Si è usata questa tecnica per compatibilità, Linux infatti supporta l'apertura delle *fifo* in lettura/scrittura, per cui si sarebbe potuto effettuare una singola apertura con `O_RDWR`; la doppia apertura comunque ha il vantaggio che non si può scrivere per errore sul capo aperto in sola lettura.

Per questo motivo, dopo aver eseguito l'apertura in lettura (24-28),⁵ si esegue una seconda apertura in scrittura (29-32), scartando il relativo file descriptor, che non sarà mai usato, in questo modo però la *fifo* resta comunque aperta anche in scrittura, cosicché le successive chiamate a `read` possono bloccarsi.

A questo punto si può entrare nel ciclo principale del programma che fornisce le risposte ai client (34-50); questo viene eseguito indefinitamente (l'uscita del server viene effettuata inviando un segnale, in modo da passare attraverso la funzione di chiusura che cancella la *fifo*).

Il server è progettato per accettare come richieste dai client delle stringhe che contengono il nome della *fifo* sulla quale deve essere inviata la risposta. Per cui prima (35-39) si esegue la lettura dalla stringa di richiesta dalla *fifo* nota (che a questo punto si bloccherà tutte le volte che non ci sono richieste). Dopo di che, una volta terminata la stringa (40) e selezionato (41) un numero casuale per ricavare la frase da inviare, si procederà (42-46) all'apertura della *fifo*

⁵di solito si effettua l'apertura del capo in lettura di una *fifo* in modalità non bloccante, per evitare il rischio di uno stallo: se infatti nessuno apre la *fifo* in scrittura il processo non ritornerà mai dalla *open*. Nel nostro caso questo rischio non esiste, mentre è necessario potersi bloccare in lettura in attesa di una richiesta.

per la risposta, che poi (47-48) vi sarà scritta. Infine (49) si chiude la *fifo* di risposta che non serve più.

Il codice del client è invece riportato in fig. 11.8, anche in questo caso si è omessa la gestione delle opzioni e la funzione che stampa a video le informazioni di utilizzo ed esce, riportando solo la sezione principale del programma e le definizioni delle variabili. Il codice completo è nel file `FortuneClient.c` dei sorgenti allegati.

```

1 int main(int argc, char *argv[])
2 {
3     /* Variables definition */
4     int n = 0;
5     char *fortunefilename = "/tmp/fortune.fifo";
6     char line[80];
7     int fifo_server, fifo_client;
8     char fifoname[80];
9     int nread;
10    char buffer[PIPE_BUF];
11    ...
12    snprintf(fifoname, 80, "/tmp/fortune.%d", getpid()); /* compose name */
13    if (mkfifo(fifoname, 0622)) { /* open client fifo */
14        if (errno!=EEXIST) {
15            perror("Cannot_create_well_known_fifo");
16            exit(-1);
17        }
18    }
19    fifo_server = open(fortunefilename, O_WRONLY); /* open server fifo */
20    if (fifo_server < 0) {
21        perror("Cannot_open_well_known_fifo");
22        exit(-1);
23    }
24    nread = write(fifo_server, fifoname, strlen(fifoname)+1); /* write name */
25    close(fifo_server); /* close server fifo */
26    fifo_client = open(fifoname, O_RDONLY); /* open client fifo */
27    if (fifo_client < 0) {
28        perror("Cannot_open_well_known_fifo");
29        exit(-1);
30    }
31    nread = read(fifo_client, buffer, sizeof(buffer)); /* read answer */
32    printf("%s", buffer); /* print fortune */
33    close(fifo_client); /* close client */
34    close(fifo_server); /* close server */
35    unlink(fifoname); /* remove client fifo */
36 }

```

Figura 11.8: Sezione principale del codice del client di *fortunes* basato sulle *fifo*.

La prima istruzione (12) compone il nome della *fifo* che dovrà essere utilizzata per ricevere la risposta dal server. Si usa il *PID* del processo per essere sicuri di avere un nome univoco; dopo di che (13-18) si procede alla creazione del relativo file, uscendo in caso di errore (a meno che il file non sia già presente sul filesystem).

A questo punto il client può effettuare l'interrogazione del server, per questo prima si apre la *fifo* nota (19-23), e poi ci si scrive (24) la stringa composta in precedenza, che contiene il nome della *fifo* da utilizzare per la risposta. Infine si richiude la *fifo* del server che a questo punto non serve più (25).

Inoltrata la richiesta si può passare alla lettura della risposta; anzitutto si apre (26-30) la *fifo* appena creata, da cui si deve riceverla, dopo di che si effettua una lettura (31) nell'apposito buffer; si è supposto, come è ragionevole, che le frasi inviate dal server siano sempre di dimensioni inferiori a PIPE_BUF, tralasciamo la gestione del caso in cui questo non è vero. Infine si stampa (32) a video la risposta, si chiude (33) la *fifo* e si cancella (34) il relativo file. Si noti come la *fifo* per la risposta sia stata aperta solo dopo aver inviato la richiesta, se non si fosse fatto così si avrebbe avuto uno stallo, in quanto senza la richiesta, il server non avrebbe potuto aprirne il capo in scrittura e l'apertura si sarebbe bloccata indefinitamente.

Verifichiamo allora il comportamento dei nostri programmi, in questo, come in altri esempi precedenti, si fa uso delle varie funzioni di servizio, che sono state raccolte nella libreria `libgapil.so`, e per poterla usare occorrerà definire la variabile di ambiente `LD_LIBRARY_PATH` in modo che il linker dinamico possa accedervi.

In generale questa variabile indica il *pathname* della directory contenente la libreria. Nell'ipotesi (che daremo sempre per verificata) che si facciano le prove direttamente nella directory dei sorgenti (dove di norma vengono creati sia i programmi che la libreria), il comando da dare sarà `export LD_LIBRARY_PATH=./`; a questo punto potremo lanciare il server, facendogli leggere una decina di frasi, con:

```
[piccardi@gont sources]$ ./fortuned -n10
```

Avendo usato `daemon` per eseguire il server in background il comando ritornerà immediatamente, ma potremo verificare con `ps` che in effetti il programma resta un'esecuzione in background, e senza avere associato un terminale di controllo (si ricordi quanto detto in sez. 8.1.5):

```
[piccardi@gont sources]$ ps aux
...
piccardi 27489  0.0  0.0 1204 356 ?        S    01:06   0:00 ./fortuned -n10
piccardi 27492  3.0  0.1 2492 764 pts/2    R    01:08   0:00 ps aux
```

e si potrà verificare anche che in `/tmp` è stata creata la *fifo* di ascolto `fortune.fifo`. A questo punto potremo interrogare il server con il programma `client`; otterremo così:

```
[piccardi@gont sources]$ ./fortune
Linux ext2fs has been stable for a long time, now it's time to break it
-- Linuxkongreß '95 in Berlin
[piccardi@gont sources]$ ./fortune
Let's call it an accidental feature.
--Larry Wall
[piccardi@gont sources]$ ./fortune
.....  Escape the 'Gates' of Hell
`:::'
::: *           `:::  :::'
::: ::  .....  ::: ::  `:::  :::'
::: ::  ::  ::  ::  ::  ::  ::
::: :::  ::  ::  `:::  ::'  :::
.....:':  :::
-- William E. Roadcap
[piccardi@gont sources]$ ./fortune
Linux ext2fs has been stable for a long time, now it's time to break it
-- Linuxkongreß '95 in Berlin
```

e ripetendo varie volte il comando otterremo, in ordine casuale, le dieci frasi tenute in memoria dal server.

Infine per chiudere il server basterà inviargli un segnale di terminazione (ad esempio con `killall fortunited`) e potremo verificare che il gestore del segnale ha anche correttamente cancellato la *fifo* di ascolto da `/tmp`.

Benché il nostro sistema client-server funzioni, la sua struttura è piuttosto complessa e continua ad avere vari inconvenienti⁶; in generale infatti l'interfaccia delle *fifo* non è adatta a risolvere questo tipo di problemi, che possono essere affrontati in maniera più semplice ed efficace o usando i socket (che tratteremo in dettaglio a partire da cap. 14) o ricorrendo a meccanismi di comunicazione diversi, come quelli che esamineremo in seguito.

11.1.5 La funzione `socketpair`

Un meccanismo di comunicazione molto simile alle *pipe*, ma che non presenta il problema della unidirezionalità del flusso dei dati, è quello dei cosiddetti *socket locali* (o *Unix domain socket*). Tratteremo in generale i socket in cap. 14, nell'ambito dell'interfaccia che essi forniscono per la programmazione di rete, e vedremo anche (in sez. 14.2.4) come si possono utilizzare i file speciali di tipo socket, analoghi a quelli associati alle *fifo* (si rammenti sez. 1.2.3) cui si accede però attraverso quella medesima interfaccia; vale però la pena esaminare qui una modalità di uso dei socket locali che li rende sostanzialmente identici ad una *pipe* bidirezionale.

La funzione di sistema `socketpair`, introdotta da BSD ma supportata in genere da qualunque sistema che fornisca l'interfaccia dei socket ed inclusa in POSIX.1-2001, consente infatti di creare una coppia di file descriptor connessi fra loro (tramite un socket, appunto) senza dover ricorrere ad un file speciale sul filesystem. I descrittori sono del tutto analoghi a quelli che si avrebbero con una chiamata a *pipe*, con la sola differenza è che in questo caso il flusso dei dati può essere effettuato in entrambe le direzioni. Il prototipo della funzione è:

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol, int sv[2])
                                Crea una coppia di socket connessi fra loro.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

- EAFNOSUPPORT i socket locali non sono supportati.
- EOPNOTSUPP il protocollo specificato non supporta la creazione di coppie di socket.
- EPROTONOSUPPORT il protocollo specificato non è supportato.
- ed inoltre EFAULT, EMFILE e ENFILE nel loro significato generico.

La funzione restituisce in `sv` la coppia di descrittori connessi fra di loro: quello che si scrive su uno di essi sarà ripresentato in input sull'altro e viceversa. Gli argomenti `domain`, `type` e `protocol` derivano dall'interfaccia dei socket (vedi sez. 14.1.2) che è quella che fornisce il substrato per connettere i due descrittori, ma in questo caso i soli valori validi che possono essere specificati sono rispettivamente `AF_UNIX`, `SOCK_STREAM` e `0`.

A partire dal kernel 2.6.27 la funzione supporta nell'indicazione del tipo di socket anche i due flag `SOCK_NONBLOCK` e `SOCK_CLOEXEC` (trattati in sez. 14.1.4), con effetto identico agli analoghi `O_CLOEXEC` e `O_NONBLOCK` di una *open* (vedi tab. 5.4).

L'utilità di chiamare questa funzione per evitare due chiamate a *pipe* può sembrare limitata; in realtà l'utilizzo di questa funzione (e dei socket locali in generale) permette di trasmettere attraverso le linee non solo dei dati, ma anche dei file descriptor: si può cioè passare da un processo ad un altro un file descriptor, con una sorta di duplicazione dello stesso non all'interno di uno stesso processo, ma fra processi distinti (torneremo su questa funzionalità in sez. 17.2.1).

⁶Lo stesso Stevens, che esamina questa architettura in [?], nota come sia impossibile per il server sapere se un client è andato in crash, con la possibilità di far restare le *fifo* temporanee sul filesystem, di come sia necessario intercettare SIGPIPE dato che un client può terminare dopo aver fatto una richiesta, ma prima che la risposta sia inviata (cosa che nel nostro esempio non è stata fatta).

11.2 L'intercomunicazione fra processi di System V

Benché le *pipe* e le *fifo* siano ancora ampiamente usate, esse scontano il limite fondamentale che il meccanismo di comunicazione che forniscono è rigidamente sequenziale: una situazione in cui un processo scrive qualcosa che molti altri devono poter leggere non può essere implementata con una *pipe*.

Per questo nello sviluppo di System V vennero introdotti una serie di nuovi oggetti per la comunicazione fra processi ed una nuova interfaccia di programmazione, poi inclusa anche in POSIX.1-2001, che fossero in grado di garantire una maggiore flessibilità. In questa sezione esamineremo come Linux supporta quello che viene chiamato il *Sistema di comunicazione fra processi* di System V, cui da qui in avanti faremo riferimento come *SysV-IPC* (dove IPC è la sigla di *Inter-Process Communication*).

11.2.1 Considerazioni generali

La principale caratteristica del *SysV-IPC* è quella di essere basato su oggetti permanenti che risiedono nel kernel. Questi, a differenza di quanto avviene per i file descriptor, non mantengono un contatore dei riferimenti, e non vengono cancellati dal sistema una volta che non sono più in uso. Questo comporta due problemi: il primo è che, al contrario di quanto avviene per *pipe* e *fifo*, la memoria allocata per questi oggetti non viene rilasciata automaticamente quando non c'è più nessuno che li utilizzi ed essi devono essere cancellati esplicitamente, se non si vuole che restino attivi fino al riavvio del sistema. Il secondo problema è, dato che non c'è come per i file un contatore del numero di riferimenti che ne indichi l'essere in uso, che essi possono essere cancellati anche se ci sono dei processi che li stanno utilizzando, con tutte le conseguenze (ovviamente assai sgradevoli) del caso.

Un'ulteriore caratteristica negativa è che gli oggetti usati nel *SysV-IPC* vengono creati direttamente dal kernel, e sono accessibili solo specificando il relativo *identificatore*. Questo è un numero progressivo (un po' come il *PID* dei processi) che il kernel assegna a ciascuno di essi quanto vengono creati (sul procedimento di assegnazione torneremo in sez. 11.2.3). L'identificatore viene restituito dalle funzioni che creano l'oggetto, ed è quindi locale al processo che le ha eseguite. Dato che l'identificatore viene assegnato dinamicamente dal kernel non è possibile prevedere quale sarà, né utilizzare un qualche valore statico, si pone perciò il problema di come processi diversi possono accedere allo stesso oggetto.

Per risolvere il problema nella struttura `ipc_perm` che il kernel associa a ciascun oggetto, viene mantenuto anche un campo apposito che contiene anche una *chiave*, identificata da una variabile del tipo primitivo `key_t`, da specificare in fase di creazione dell'oggetto, e tramite la quale è possibile ricavare l'identificatore.⁷ Oltre la chiave, la struttura, la cui definizione è riportata in fig. 11.9, mantiene varie proprietà ed informazioni associate all'oggetto.

Usando la stessa chiave due processi diversi possono ricavare l'identificatore associato ad un oggetto ed accedervi. Il problema che sorge a questo punto è come devono fare per accordarsi sull'uso di una stessa chiave. Se i processi sono *imparentati* la soluzione è relativamente semplice, in tal caso infatti si può usare il valore speciale `IPC_PRIVATE` per creare un nuovo oggetto nel processo padre, l'identificatore così ottenuto sarà disponibile in tutti i figli, e potrà essere passato come argomento attraverso una `exec`.

Però quando i processi non sono *imparentati* (come capita tutte le volte che si ha a che fare con un sistema client-server) tutto questo non è possibile; si potrebbe comunque salvare

⁷in sostanza si sposta il problema dell'accesso dalla classificazione in base all'identificatore alla classificazione in base alla chiave, una delle tante complicazioni inutili presenti nel *SysV-IPC*.

```

struct ipc_perm
{
    key_t key;                /* Key. */
    uid_t uid;                /* Owner's user ID. */
    gid_t gid;                /* Owner's group ID. */
    uid_t cuid;               /* Creator's user ID. */
    gid_t cgid;               /* Creator's group ID. */
    unsigned short int mode;  /* Read/write permission. */
    unsigned short int seq;   /* Sequence number. */
};

```

Figura 11.9: La struttura `ipc_perm`, come definita in `sys/ipc.h`.

l'identificatore su un file noto, ma questo ovviamente comporta lo svantaggio di doverlo andare a rileggere. Una alternativa più efficace è quella che i programmi usino un valore comune per la chiave (che ad esempio può essere dichiarato in un header comune), ma c'è sempre il rischio che questa chiave possa essere stata già utilizzata da qualcun altro. Dato che non esiste una convenzione su come assegnare queste chiavi in maniera univoca l'interfaccia mette a disposizione una funzione apposita, `ftok`, che permette di ottenere una chiave specificando il nome di un file ed un numero di versione; il suo prototipo è:

<pre> #include <sys/types.h> #include <sys/ipc.h> key_t ftok(const char *pathname, int proj_id) </pre>	<p>Restituisce una chiave per identificare un oggetto del <i>SysV IPC</i>.</p>
<p>La funzione ritorna la chiave in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei possibili codici di errore di <code>stat</code>.</p>	

La funzione determina un valore della chiave sulla base di `pathname`, che deve specificare il `pathname` di un file effettivamente esistente e di un numero di progetto `proj_id`, che di norma viene specificato come carattere, dato che ne vengono utilizzati solo gli 8 bit meno significativi. Nelle *libc4* e *libc5*, come avviene in SunOS, l'argomento `proj_id` è dichiarato tipo `char`, la *glibc* usa il prototipo specificato da XPG4, ma vengono lo stesso utilizzati gli 8 bit meno significativi.

Il problema è che anche così non c'è la sicurezza che il valore della chiave sia univoco, infatti esso è costruito combinando il byte di `proj_id` con i 16 bit meno significativi dell'inode del file `pathname` (che vengono ottenuti attraverso `stat`, da cui derivano i possibili errori), e gli 8 bit meno significativi del numero del dispositivo su cui è il file. Diventa perciò relativamente facile ottenere delle collisioni, specie se i file sono su dispositivi con lo stesso *minor number*, come `/dev/hda1` e `/dev/sda1`.

In genere quello che si fa è utilizzare un file comune usato dai programmi che devono comunicare (ad esempio un header comune, o uno dei programmi che devono usare l'oggetto in questione), utilizzando il numero di progetto per ottenere le chiavi che interessano. In ogni caso occorre sempre controllare, prima di creare un oggetto, che la chiave non sia già stata utilizzata. Se questo va bene in fase di creazione, le cose possono complicarsi per i programmi che devono solo accedere, in quanto, a parte gli eventuali controlli sugli altri attributi di `ipc_perm`, non esiste una modalità semplice per essere sicuri che l'oggetto associato ad una certa chiave sia stato effettivamente creato da chi ci si aspetta.

Questo è, insieme al fatto che gli oggetti sono permanenti e non mantengono un contatore di riferimenti per la cancellazione automatica, il principale problema del *SysV-IPC*. Non

esiste infatti una modalità chiara per identificare un oggetto, come sarebbe stato se lo si fosse associato ad un file, e tutta l'interfaccia è inutilmente complessa. Per questo se ne sconsiglia assolutamente l'uso nei nuovi programmi, considerato che è ormai disponibile una revisione completa dei meccanismi di IPC fatta secondo quanto indicato dallo standard POSIX.1b, che presenta una realizzazione più sicura ed una interfaccia più semplice, che tratteremo in sez. 11.4.

11.2.2 Il controllo di accesso

Oltre alle chiavi, abbiamo visto in fig. 11.9 che ad ogni oggetto sono associate in `ipc_perm` ulteriori informazioni, come gli identificatori del creatore (nei campi `cuid` e `cgid`) e del proprietario (nei campi `uid` e `gid`) dello stesso, e un insieme di permessi (nel campo `mode`). In questo modo è possibile definire un controllo di accesso sugli oggetti di IPC, simile a quello che si ha per i file (vedi sez. 4.4.1).

Benché questo controllo di accesso sia molto simile a quello dei file, restano delle importanti differenze. La prima è che il permesso di esecuzione non esiste (e se specificato viene ignorato), per cui si può parlare solo di permessi di lettura e scrittura (nel caso dei semafori poi quest'ultimo è più propriamente un permesso di modifica). I valori di `mode` sono gli stessi ed hanno lo stesso significato di quelli riportati in tab. 4.7 e come per i file definiscono gli accessi per il proprietario, il suo gruppo e tutti gli altri.

Se però si vogliono usare le costanti simboliche di tab. 4.7 occorrerà includere anche il file `sys/stat.h`; alcuni sistemi definiscono le costanti `MSG_R` (il valore ottale `0400`) e `MSG_W` (il valore ottale `0200`) per indicare i permessi base di lettura e scrittura per il proprietario, da utilizzare, con gli opportuni shift, pure per il gruppo e gli altri. In Linux, visto la loro scarsa utilità, queste costanti non sono definite.

Quando l'oggetto viene creato i campi `cuid` e `uid` di `ipc_perm` ed i campi `cgid` e `gid` vengono impostati rispettivamente al valore dell'`UID` e del `GID` effettivo del processo che ha chiamato la funzione, ma, mentre i campi `uid` e `gid` possono essere cambiati, i campi `cuid` e `cgid` restano sempre gli stessi.

Il controllo di accesso è effettuato a due livelli. Il primo livello è nelle funzioni che richiedono l'identificatore di un oggetto data la chiave. Queste specificano tutte un argomento `flag`, in tal caso quando viene effettuata la ricerca di una chiave, qualora `flag` specifichi dei permessi, questi vengono controllati e l'identificatore viene restituito solo se corrispondono a quelli dell'oggetto. Se ci sono dei permessi non presenti in `mode` l'accesso sarà negato. Questo controllo però è di utilità indicativa, dato che è sempre possibile specificare per `flag` un valore nullo, nel qual caso l'identificatore sarà restituito comunque.

Il secondo livello di controllo è quello delle varie funzioni che accedono direttamente (in lettura o scrittura) all'oggetto. In tal caso lo schema dei controlli è simile a quello dei file, ed avviene secondo questa sequenza:

- se il processo ha i privilegi di amministratore (più precisamente `CAP_IPC_OWNER`) l'accesso è sempre consentito.
- se l'`UID` effettivo del processo corrisponde o al valore del campo `cuid` o a quello del campo `uid` ed il permesso per il proprietario in `mode` è appropriato⁸ l'accesso è consentito.
- se il `GID` effettivo del processo corrisponde o al valore del campo `cgid` o a quello del campo `gid` ed il permesso per il gruppo in `mode` è appropriato l'accesso è consentito.
- se il permesso per gli altri è appropriato l'accesso è consentito.

⁸per appropriato si intende che è impostato il permesso di scrittura per le operazioni di scrittura e quello di lettura per le operazioni di lettura.

solo se tutti i controlli elencati falliscono l'accesso è negato. Si noti che a differenza di quanto avviene per i permessi dei file, fallire in uno dei passi elencati non comporta il fallimento dell'accesso. Un'ulteriore differenza rispetto a quanto avviene per i file è che per gli oggetti di IPC il valore di *umask* (si ricordi quanto esposto in sez. 4.4.3) non ha alcun significato.

11.2.3 Gli identificatori ed il loro utilizzo

L'unico campo di *ipc_perm* del quale non abbiamo ancora parlato è *seq*, che in fig. 11.9 è qualificato con un criptico “*numero di sequenza*”, ne parliamo adesso dato che esso è strettamente attinente alle modalità con cui il kernel assegna gli identificatori degli oggetti del sistema di IPC.

Quando il sistema si avvia, alla creazione di ogni nuovo oggetto di IPC viene assegnato un numero progressivo, pari al numero di oggetti di quel tipo esistenti. Se il comportamento fosse sempre questo sarebbe identico a quello usato nell'assegnazione dei file descriptor nei processi, ed i valori degli identificatori tenderebbero ad essere riutilizzati spesso e restare di piccole dimensioni (inferiori al numero massimo di oggetti disponibili).

Questo va benissimo nel caso dei file descriptor, che sono locali ad un processo, ma qui il comportamento varrebbe per tutto il sistema, e per processi del tutto scorrelati fra loro. Così si potrebbero avere situazioni come quella in cui un server esce e cancella le sue code di messaggi, ed il relativo identificatore viene immediatamente assegnato a quelle di un altro server partito subito dopo, con la possibilità che i client del primo non facciano in tempo ad accorgersi dell'avvenuto, e finiscano con l'interagire con gli oggetti del secondo, con conseguenze imprevedibili.

Proprio per evitare questo tipo di situazioni il sistema usa il valore di *seq* per provvedere un meccanismo che porti gli identificatori ad assumere tutti i valori possibili, rendendo molto più lungo il periodo in cui un identificatore può venire riutilizzato.

Il sistema dispone sempre di un numero fisso di oggetti di IPC, fino al kernel 2.2.x questi erano definiti dalle costanti *MSGMNI*, *SEMMNI* e *SHMMNI*, e potevano essere cambiati (come tutti gli altri limiti relativi al *SysV-IPC*) solo con una ricompilazione del kernel. A partire dal kernel 2.4.x è possibile cambiare questi valori a sistema attivo scrivendo sui file *shmmni*, *msgmni* e *sem* di */proc/sys/kernel* o con l'uso di *sysctl*.

Per ciascun tipo di oggetto di IPC viene mantenuto in *seq* un numero di sequenza progressivo che viene incrementato di uno ogni volta che l'oggetto viene cancellato. Quando l'oggetto viene creato usando uno spazio che era già stato utilizzato in precedenza, per restituire il nuovo identificatore al numero di oggetti presenti viene sommato il valore corrente del campo *seq*, moltiplicato per il numero massimo di oggetti di quel tipo.

Questo in realtà è quanto avveniva fino al kernel della serie 2.2, dalla serie 2.4 viene usato lo stesso fattore di moltiplicazione per qualunque tipo di oggetto, utilizzando il valore dalla costante *IPCMMNI* (definita in *include/linux/ipc.h*), che indica il limite massimo complessivo per il numero di tutti gli oggetti presenti nel *SysV-IPC*, ed il cui default è 32768. Si evita così il riutilizzo degli stessi numeri, e si fa sì che l'identificatore assuma tutti i valori possibili.

In fig. 11.10 è riportato il codice di un semplice programma di test che si limita a creare un oggetto di IPC (specificato con una opzione a riga di comando), stamparne il numero di identificatore, e cancellarlo, il tutto un numero di volte specificato tramite una seconda opzione. La figura non riporta il codice di selezione delle opzioni, che permette di inizializzare i valori delle variabili *type* al tipo di oggetto voluto, e *n* al numero di volte che si vuole effettuare il ciclo di creazione, stampa, cancellazione.

I valori di default sono per l'uso delle code di messaggi e per 5 ripetizioni del ciclo. Per questo motivo se non si utilizzano opzioni verrà eseguito per cinque volte il ciclo (7-11), in cui

```

1 int main(int argc, char *argv[])
2 {
3     ...
4     switch (type) {
5     case 'q': /* Message Queue */
6         debug("Message_Queue_Try\n");
7         for (i=0; i<n; i++) {
8             id = msgget(IPC_PRIVATE, IPC_CREAT|0666);
9             printf("Identifier_Value_%d\n", id);
10            msgctl(id, IPC_RMID, NULL);
11        }
12        break;
13     case 's': /* Semaphore */
14        debug("Semaphore\n");
15        for (i=0; i<n; i++) {
16            id = semget(IPC_PRIVATE, 1, IPC_CREAT|0666);
17            printf("Identifier_Value_%d\n", id);
18            semctl(id, 0, IPC_RMID);
19        }
20        break;
21     case 'm': /* Shared Memory */
22        debug("Shared_Memory\n");
23        for (i=0; i<n; i++) {
24            id = shmget(IPC_PRIVATE, 1000, IPC_CREAT|0666);
25            printf("Identifier_Value_%d\n", id);
26            shmctl(id, IPC_RMID, NULL);
27        }
28        break;
29     default: /* should not reached */
30        return -1;
31    }
32    return 0;
33 }

```

Figura 11.10: Sezione principale del programma di test per l'assegnazione degli identificatori degli oggetti di IPC IPCTestId.c.

si crea una coda di messaggi (8), se ne stampa l'identificativo (9) e la si rimuove (10). Non stiamo ad approfondire adesso il significato delle funzioni utilizzate, che verranno esaminate nelle prossime sezioni.

Quello che ci interessa infatti è verificare l'allocazione degli identificativi associati agli oggetti; lanciando il comando si otterrà pertanto qualcosa del tipo:

```

piccardi@gont sources]$ ./ipctestid
Identifier Value 0
Identifier Value 32768
Identifier Value 65536
Identifier Value 98304
Identifier Value 131072

```

il che ci mostra che stiamo lavorando con un kernel posteriore alla serie 2.2 nel quale non avevamo ancora usato nessuna coda di messaggi (il valore nullo del primo identificativo indica che il campo `seq` era zero). Ripetendo il comando, e quindi eseguendolo in un processo diverso, in cui non può esistere nessuna traccia di quanto avvenuto in precedenza, otterremo come nuovo risultato:

```
[piccardi@gont sources]$ ./ipctestid
Identifier Value 163840
Identifier Value 196608
Identifier Value 229376
Identifier Value 262144
Identifier Value 294912
```

in cui la sequenza numerica prosegue, cosa che ci mostra come il valore di `seq` continui ad essere incrementato e costituisca in effetti una quantità mantenuta all'interno del sistema ed indipendente dai processi.

11.2.4 Code di messaggi

Il primo oggetto introdotto dal *SysV-IPC* è quello delle code di messaggi. Le code di messaggi sono oggetti analoghi alle *pipe* o alle *fifo* ed il loro scopo principale è quello di fornire a processi diversi un meccanismo con cui scambiarsi dei dati in forma di messaggio. Dato che le *pipe* e le *fifo* costituiscono una ottima alternativa, ed in genere sono molto più semplici da usare, le code di messaggi sono il meno utilizzato degli oggetti introdotti dal *SysV-IPC*.

La funzione di sistema che permette di ottenere l'identificativo di una coda di messaggi esistente per potervi accedere, oppure di creare una nuova coda qualora quella indicata non esista ancora, è `msgget`, e il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int flag)
```

Ottiene o crea una coda di messaggi.

La funzione ritorna l'identificatore (un intero positivo) in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES il processo chiamante non ha i privilegi per accedere alla coda richiesta.

EEXIST si è richiesta la creazione di una coda che già esiste, ma erano specificati sia `IPC_CREAT` che `IPC_EXCL`.

EIDRM la coda richiesta è marcata per essere cancellata (solo fino al kernel 2.3.20).

ENOENT si è cercato di ottenere l'identificatore di una coda di messaggi specificando una chiave che non esiste e `IPC_CREAT` non era specificato.

ENOSPC si è cercato di creare una coda di messaggi quando è stato superato il limite massimo di code (`MSGMNI`).

ed inoltre `ENOMEM` nel suo significato generico.

Le funzione (come le analoghe che si usano per gli altri oggetti) serve sia a ottenere l'identificatore di una coda di messaggi esistente, che a crearne una nuova. L'argomento `key` specifica la chiave che è associata all'oggetto, eccetto il caso in cui si specifichi il valore `IPC_PRIVATE`, nel qual caso la coda è creata *ex-novo* e non vi è associata alcuna chiave (per questo viene detta *privata*), ed il processo e i suoi eventuali figli potranno farvi riferimento solo attraverso l'identificatore.

Se invece si specifica un valore diverso da `IPC_PRIVATE` (in Linux questo significa un valore diverso da zero) l'effetto della funzione dipende dal valore di `flag`, se questo è nullo la funzione si limita ad effettuare una ricerca sugli oggetti esistenti, restituendo l'identificatore se trova una corrispondenza, o fallendo con un errore di `ENOENT` se non esiste o di `EACCES` se si sono specificati dei permessi non validi.

Se invece si vuole creare una nuova coda di messaggi `flag` non può essere nullo e deve essere fornito come maschera binaria, impostando il bit corrispondente al valore `IPC_CREAT`. In questo caso i nove bit meno significativi di `flag` saranno usati come permessi per il nuovo

oggetto, secondo quanto illustrato in sez. 11.2.2. Se si imposta anche il bit corrispondente a `IPC_EXCL` la funzione avrà successo solo se l'oggetto non esiste già, fallendo con un errore di `EEXIST` altrimenti.

Si tenga conto che l'uso di `IPC_PRIVATE` non impedisce ad altri processi di accedere alla coda, se hanno privilegi sufficienti, una volta che questi possano indovinare o ricavare, ad esempio per tentativi, l'identificatore ad essa associato. Per come sono implementati gli oggetti di IPC infatti non esiste alcun modo in cui si possa garantire l'accesso esclusivo ad una coda di messaggi. Usare `IPC_PRIVATE` o `IPC_CREAT` e `IPC_EXCL` per `flag` comporta solo la creazione di una nuova coda.

Costante	Valore	File in <code>/proc</code>	Significato
<code>MSGMNI</code>	16	<code>msgmni</code>	Numero massimo di code di messaggi.
<code>MSGMAX</code>	8192	<code>msgmax</code>	Dimensione massima di un singolo messaggio.
<code>MSGMB</code>	16384	<code>msgmb</code>	Dimensione massima del contenuto di una coda.

Tabella 11.1: Valori delle costanti associate ai limiti delle code di messaggi.

Le code di messaggi sono caratterizzate da tre limiti fondamentali, un tempo definiti staticamente e corrispondenti alle prime tre costanti riportate in tab. 11.1. Come accennato però con tutte le versioni più recenti del kernel con Linux è possibile modificare questi limiti attraverso l'uso di `sysctl` o scrivendo nei file `msgmax`, `msgmb` e `msgmni` di `/proc/sys/kernel/`.

Una coda di messaggi è costituita da una *linked list*.⁹ I nuovi messaggi vengono inseriti in coda alla lista e vengono letti dalla cima, in fig. 11.11 si è riportato uno schema semplificato con cui queste strutture vengono mantenute dal kernel. Lo schema illustrato in realtà è una semplificazione di quello usato fino ai kernel della serie 2.2. A partire della serie 2.4 la gestione delle code di messaggi è effettuata in maniera diversa (e non esiste una struttura `msqid_ds` nel kernel), ma abbiamo mantenuto lo schema precedente dato che illustra in maniera più che adeguata i principi di funzionamento delle code di messaggi.

Figura 11.11: Schema delle strutture di una coda di messaggi (`msqid_ds` e `msg`).

A ciascuna coda è associata una struttura `msqid_ds` la cui definizione è riportata in fig. 11.12 ed a cui si accede includendo `sys/msg.h`; si tenga presente che il campo `__msg_cbytes` non è previsto dallo standard POSIX.1-2001 e che alcuni campi fino al kernel 2.2 erano definiti come `short`.

Quando si crea una nuova coda con `msgget` questa struttura viene inizializzata,¹⁰ in particolare il campo `msg_perm` che esprime i permessi di accesso viene inizializzato nella modalità illustrata in sez. 11.2.2. Per quanto riguarda gli altri campi invece:

- il campo `msg_qnum`, che esprime il numero di messaggi presenti sulla coda, viene inizializzato a 0.
- i campi `msg_lspid` e `msg_lrpid`, che esprimono rispettivamente il *PID* dell'ultimo processo che ha inviato o ricevuto un messaggio sulla coda, sono inizializzati a 0.
- i campi `msg_stime` e `msg_rtime`, che esprimono rispettivamente il tempo in cui è stato inviato o ricevuto l'ultimo messaggio sulla coda, sono inizializzati a 0.

⁹una *linked list* è una tipica struttura di dati, organizzati in una lista in cui ciascun elemento contiene un puntatore al successivo. In questo modo la struttura è veloce nell'estrazione ed immissione dei dati dalle estremità dalla lista (basta aggiungere un elemento in testa o in coda ed aggiornare un puntatore), e relativamente veloce da attraversare in ordine sequenziale (seguendo i puntatori), è invece relativamente lenta nell'accesso casuale e nella ricerca.

¹⁰in realtà viene inizializzata una struttura interna al kernel, ma i dati citati sono gli stessi.

```

struct msqid_ds {
    struct ipc_perm msg_perm;    /* structure for operation permission */
    time_t msg_stime;           /* time of last msgsnd command */
    time_t msg_rtime;           /* time of last msgrcv command */
    time_t msg_ctime;           /* time of last change */
    unsigned long __msg_cbytes; /* current number of bytes on queue */
    msgqnum_t msg_qnum;         /* number of messages currently on queue */
    msglen_t msg_qbytes;        /* max number of bytes allowed on queue */
    pid_t msg_lspid;            /* pid of last msgsnd() */
    pid_t msg_lrpid;            /* pid of last msgrcv() */
};

```

Figura 11.12: La struttura `msqid_ds`, associata a ciascuna coda di messaggi.

- il campo `msg_ctime`, che esprime il tempo di ultima modifica della coda, viene inizializzato al tempo corrente.
- il campo `msg_qbytes`, che esprime la dimensione massima del contenuto della coda (in byte) viene inizializzato al valore preimpostato del sistema (`MSGMNB`).
- il campo `__msg_cbytes`, che esprime la dimensione in byte dei messaggi presenti sulla coda, viene inizializzato a zero.

Una volta creata una coda di messaggi le operazioni di controllo vengono effettuate con la funzione di sistema `msgctl`, che, come le analoghe `semctl` e `shmctl`, fa le veci di quello che `ioctl` è per i file; il suo prototipo è:

```

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf)

```

Esegue una operazione su una coda.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES	si è richiesto IPC_STAT ma processo chiamante non ha i privilegi di lettura sulla coda.
EIDRM	la coda richiesta è stata cancellata.
EPERM	si è richiesto IPC_SET o IPC_RMID ma il processo non ha i privilegi, o si è richiesto di aumentare il valore di <code>msg_qbytes</code> oltre il limite <code>MSGMNB</code> senza essere amministratore.

ed inoltre EFAULT ed EINVAL nel loro significato generico.

La funzione permette di eseguire una operazione di controllo per la coda specificata dall'identificatore `msqid`, utilizzando i valori della struttura `msqid_ds`, mantenuta all'indirizzo `buf`. Il comportamento della funzione dipende dal valore dell'argomento `cmd`, che specifica il tipo di azione da eseguire. I valori possibili per `cmd` sono:

- IPC_STAT** Legge le informazioni riguardo la coda nella struttura `msqid_ds` indicata da `buf`. Occorre avere il permesso di lettura sulla coda.
- IPC_RMID** Rimuove la coda, cancellando tutti i dati, con effetto immediato. Tutti i processi che cercheranno di accedere alla coda riceveranno un errore di `EIDRM`, e tutti processi in attesa su funzioni di lettura o di scrittura sulla coda saranno svegliati ricevendo il medesimo errore. Questo comando può essere eseguito solo da un processo con *UID* effettivo corrispondente al creatore o al proprietario della coda, o all'amministratore.

IPC_SET Permette di modificare i permessi ed il proprietario della coda, ed il limite massimo sulle dimensioni del totale dei messaggi in essa contenuti (`msg_qbytes`). I valori devono essere passati in una struttura `msqid_ds` puntata da `buf`. Per modificare i valori di `msg_perm.mode`, `msg_perm.uid` e `msg_perm.gid` occorre essere il proprietario o il creatore della coda, oppure l'amministratore e lo stesso vale per `msg_qbytes`. Infine solo l'amministratore (più precisamente un processo con la capacità `CAP_IPC_RESOURCE`) ha la facoltà di incrementarne il valore a limiti superiori a `MSGMNB`. Se eseguita con successo la funzione aggiorna anche il campo `msg_ctime`.

A questi tre valori, che sono quelli previsti dallo standard, su Linux se ne affiancano altri tre (`IPC_INFO`, `MSG_STAT` e `MSG_INFO`) introdotti ad uso del programma `ipcs` per ottenere le informazioni generali relative alle risorse usate dalle code di messaggi. Questi potranno essere modificati o rimossi in favore dell'uso di `/proc`, per cui non devono essere usati e non li tratteremo.

Una volta che si abbia a disposizione l'identificatore, per inviare un messaggio su una coda si utilizza la funzione di sistema `msgsnd`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
```

Invia un messaggio su una coda.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES non si hanno i privilegi di accesso sulla coda.

EAGAIN il messaggio non può essere inviato perché si è superato il limite `msg_qbytes` sul numero massimo di byte presenti sulla coda, e si è richiesto `IPC_NOWAIT` in `flag`.

EIDRM la coda è stata cancellata.

EINVAL si è specificato un `msgid` invalido, o un valore non positivo per `mtype`, o un valore di `msgsz` maggiore di `MSGMAX`.

ed inoltre `EFAULT`, `EINTR` e `ENOMEM` nel loro significato generico.

La funzione inserisce il messaggio sulla coda specificata da `msqid`; il messaggio ha lunghezza specificata da `msgsz` ed è passato attraverso il l'argomento `msgp`. Quest'ultimo deve venire passato sempre come puntatore ad una struttura `msgbuf` analoga a quella riportata in fig. 11.13 che è quella che deve contenere effettivamente il messaggio. La dimensione massima per il testo di un messaggio non può comunque superare il limite `MSGMAX`.

```
struct msgbuf {
    long mtype;           /* message type, must be > 0 */
    char mtext[LENGTH];  /* message data */
};
```

Figura 11.13: Schema della struttura `msgbuf`, da utilizzare come argomento per inviare/ricevere messaggi.

La struttura di fig. 11.13 è comunque solo un modello, tanto che la definizione contenuta in `sys/msg.h` usa esplicitamente per il secondo campo il valore `mtext[1]`, che non è di nessuna utilità ai fini pratici. La sola cosa che conta è che la struttura abbia come primo membro un campo `mtype` come nell'esempio; esso infatti serve ad identificare il tipo di messaggio e deve

essere sempre specificato come intero positivo di tipo `long`. Il campo `mtext` invece può essere di qualsiasi tipo e dimensione, e serve a contenere il testo del messaggio.

In generale pertanto per inviare un messaggio con `msgsnd` si usa ridefinire una struttura simile a quella di fig. 11.13, adattando alle proprie esigenze il campo `mtype`, (o ridefinendo come si vuole il corpo del messaggio, anche con più campi o con strutture più complesse) avendo però la cura di mantenere nel primo campo un valore di tipo `long` che ne indica il tipo.

Si tenga presente che la lunghezza che deve essere indicata in questo argomento è solo quella del messaggio, non quella di tutta la struttura, se cioè `message` è una propria struttura che si passa alla funzione, `msgsz` dovrà essere uguale a `sizeof(message)-sizeof(long)`, (se consideriamo il caso dell'esempio in fig. 11.13, `msgsz` dovrà essere pari a `LENGTH`).

Per capire meglio il funzionamento della funzione riprendiamo in considerazione la struttura della coda illustrata in fig. 11.11. Alla chiamata di `msgsnd` il nuovo messaggio sarà aggiunto in fondo alla lista inserendo una nuova struttura `msg`, il puntatore `msg_last` di `msqid_ds` verrà aggiornato, come pure il puntatore al messaggio successivo per quello che era il precedente ultimo messaggio; il valore di `mtype` verrà mantenuto in `msg_type` ed il valore di `msgsz` in `msg_ts`; il testo del messaggio sarà copiato all'indirizzo specificato da `msg_spot`.

Il valore dell'argomento `flag` permette di specificare il comportamento della funzione. Di norma, quando si specifica un valore nullo, la funzione ritorna immediatamente a meno che si sia ecceduto il valore di `msg_qbytes`, o il limite di sistema sul numero di messaggi, nel qual caso si blocca. Se si specifica per `flag` il valore `IPC_NOWAIT` la funzione opera in modalità non-bloccante, ed in questi casi ritorna immediatamente con un errore di `EAGAIN`.

Se non si specifica `IPC_NOWAIT` la funzione resterà bloccata fintanto che non si liberano risorse sufficienti per poter inserire nella coda il messaggio, nel qual caso ritornerà normalmente. La funzione può ritornare con una condizione di errore anche in due altri casi: quando la coda viene rimossa (nel qual caso si ha un errore di `EIDRM`) o quando la funzione viene interrotta da un segnale (nel qual caso si ha un errore di `EINTR`).

Una volta completato con successo l'invio del messaggio sulla coda, la funzione aggiorna i dati mantenuti in `msqid_ds`, in particolare vengono modificati:

- Il valore di `msg_lspid`, che viene impostato al *PID* del processo chiamante.
- Il valore di `msg_qnum`, che viene incrementato di uno.
- Il valore `msg_stime`, che viene impostato al tempo corrente.

La funzione di sistema che viene utilizzata per estrarre un messaggio da una coda è `msgrcv`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg)
Legge un messaggio da una coda.
```

La funzione ritorna il numero di byte letti in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

<code>E2BIG</code>	il testo del messaggio è più lungo di <code>msgsz</code> e non si è specificato <code>MSG_NOERROR</code> in <code>msgflg</code> .
<code>EACCES</code>	non si hanno i privilegi di accesso sulla coda.
<code>EIDRM</code>	la coda è stata cancellata.
<code>EINTR</code>	la funzione è stata interrotta da un segnale mentre era in attesa di ricevere un messaggio.
<code>EINVAL</code>	si è specificato un <code>msgid</code> invalido o un valore di <code>msgsz</code> negativo.

ed inoltre `EFAULT` nel suo significato generico.

La funzione legge un messaggio dalla coda specificata da `msqid`, scrivendolo sulla struttura puntata da `msgp`, che dovrà avere un formato analogo a quello di fig. 11.13. Una volta estratto, il messaggio sarà rimosso dalla coda. L'argomento `msgsz` indica la lunghezza massima del testo del messaggio (equivalente al valore del parametro `LENGTH` nell'esempio di fig. 11.13).

Se il testo del messaggio ha lunghezza inferiore a `msgsz` esso viene rimosso dalla coda; in caso contrario, se `msgflg` è impostato a `MSG_NOERROR`, il messaggio viene troncato e la parte in eccesso viene perduta, altrimenti il messaggio non viene estratto e la funzione ritorna con un errore di `E2BIG`.

L'argomento `msgtyp` permette di restringere la ricerca ad un sottoinsieme dei messaggi presenti sulla coda; la ricerca infatti è fatta con una scansione della struttura mostrata in fig. 11.11, restituendo il primo messaggio incontrato che corrisponde ai criteri specificati (che quindi, visto come i messaggi vengono sempre inseriti dalla coda, è quello meno recente); in particolare:

- se `msgtyp` è 0 viene estratto il messaggio in cima alla coda, cioè quello fra i presenti che è stato inserito per primo.
- se `msgtyp` è positivo viene estratto il primo messaggio il cui tipo (il valore del campo `mtype`) corrisponde al valore di `msgtyp`.
- se `msgtyp` è negativo viene estratto il primo fra i messaggi con il valore più basso del tipo, fra tutti quelli il cui tipo ha un valore inferiore al valore assoluto di `msgtyp`.

Il valore di `msgflg` permette di controllare il comportamento della funzione, esso può essere nullo o una maschera binaria composta da uno o più valori. Oltre al precedente `MSG_NOERROR`, sono possibili altri due valori: `MSG_EXCEPT`, che permette, quando `msgtyp` è positivo, di leggere il primo messaggio nella coda con tipo diverso da `msgtyp`, e `IPC_NOWAIT` che causa il ritorno immediato della funzione quando non ci sono messaggi sulla coda.

Il comportamento usuale della funzione infatti, se non ci sono messaggi disponibili per la lettura, è di bloccare il processo. Nel caso però si sia specificato `IPC_NOWAIT` la funzione ritorna immediatamente con un errore `ENOMSG`. Altrimenti la funzione ritorna normalmente non appena viene inserito un messaggio del tipo desiderato, oppure ritorna con errore qualora la coda sia rimossa (con `errno` impostata a `EIDRM`) o se il processo viene interrotto da un segnale (con `errno` impostata a `EINTR`).

Una volta completata con successo l'estrazione del messaggio dalla coda, la funzione aggiorna i dati mantenuti in `msqid_ds`, in particolare vengono modificati:

- Il valore di `msg_lrpids`, che viene impostato al *PID* del processo chiamante.
- Il valore di `msg_qnum`, che viene decrementato di uno.
- Il valore `msg_rtime`, che viene impostato al tempo corrente.

Le code di messaggi presentano il solito problema di tutti gli oggetti del *SysV-IPC* che essendo questi permanenti restano nel sistema occupando risorse anche quando un processo è terminato, al contrario delle *pipe* per le quali tutte le risorse occupate vengono rilasciate quanto l'ultimo processo che le utilizzava termina. Questo comporta che in caso di errori si può saturare il sistema, e che devono comunque essere esplicitamente previste delle funzioni di rimozione in caso di interruzioni o uscite dal programma (come vedremo in fig. 11.14).

L'altro problema è che non facendo uso di file descriptor le tecniche di *I/O multiplexing* descritte in sez. 10.2 non possono essere utilizzate, e non si ha a disposizione niente di analogo alle funzioni `select` e `poll`. Questo rende molto scomodo usare più di una di queste strutture alla volta; ad esempio non si può scrivere un server che aspetti un messaggio su più di una

coda senza fare ricorso ad una tecnica di *polling* che esegua un ciclo di attesa su ciascuna di esse.

Come esempio dell'uso delle code di messaggi possiamo riscrivere il nostro server di *fortunes* usando queste al posto delle *fifo*. In questo caso useremo una sola coda di messaggi, usando il tipo di messaggio per comunicare in maniera indipendente con client diversi.

```

1 int msgid;           /* Message queue identifier */
2 int main(int argc, char *argv[])
3 {
4     /* Variables definition */
5     int i, n = 0;
6     char **fortune;    /* array of fortune message string */
7     char *fortunefilename = "/usr/share/games/fortunes/linux"; /* file name */
8     struct msgbuf_read { /* message struct to read request from clients */
9         long mtype;      /* message type, must be 1 */
10        long pid;         /* message data, must be the pid of the client */
11    } msg_read;
12    struct msgbuf_write { /* message struct to write result to clients */
13        long mtype;      /* message type, will be the pid of the client */
14        char mtext[MSGMAX]; /* message data, will be the fortune */
15    } msg_write;
16    key_t key;          /* Message queue key */
17    int size;           /* message size */
18    ...
19    Signal(SIGTERM, HandSIGTERM); /* set handlers for termination */
20    Signal(SIGINT, HandSIGTERM);
21    Signal(SIGQUIT, HandSIGTERM);
22    if (n==0) usage(); /* if no pool depth exit printing usage info */
23    i = FortuneParse(fortunefilename, fortune, n); /* parse phrases */
24    /* Create the queue */
25    key = ftok("./MQFortuneServer.c", 1);
26    msgid = msgget(key, IPC_CREAT|0666);
27    if (msgid < 0) {
28        perror("Cannot create message queue");
29        exit(1);
30    }
31    /* Main body: loop over requests */
32    daemon(0, 0);
33    while (1) {
34        msgrcv(msgid, &msg_read, sizeof(int), 1, MSG_NOERROR);
35        n = random() % i; /* select random value */
36        strncpy(msg_write.mtext, fortune[n], MSGMAX);
37        size = min(strlen(fortune[n])+1, MSGMAX);
38        msg_write.mtype=msg_read.pid; /* use request pid as type */
39        msgsnd(msgid, &msg_write, size, 0);
40    }
41 }
42 /*
43  * Signal Handler to manage termination
44  */
45 void HandSIGTERM(int signo) {
46     msgctl(msgid, IPC_RMID, NULL); /* remove message queue */
47     exit(0);
48 }

```

Figura 11.14: Sezione principale del codice del server di *fortunes* basato sulle *message queue*.

In fig. 11.14 si è riportato un estratto delle parti principali del codice del nuovo server (il codice completo è nel file `MQFortuneServer.c` nei sorgenti allegati). Il programma è basato su un uso accorto della caratteristica di poter associare un “tipo” ai messaggi per permettere una comunicazione indipendente fra il server ed i vari client, usando il *PID* di questi ultimi come identificativo. Questo è possibile in quanto, al contrario di una *fifo*, la lettura di una coda di messaggi può non essere sequenziale, proprio grazie alla classificazione dei messaggi sulla base del loro tipo.

Il programma, oltre alle solite variabili per il nome del file da cui leggere le *fortunes* e per il vettore di stringhe che contiene le frasi, definisce due strutture appositamente per la comunicazione; con `msgbuf_read` vengono passate (8-11) le richieste mentre con `msgbuf_write` vengono restituite (12-15) le frasi.

La gestione delle opzioni si è al solito omessa, essa si occuperà di impostare nella variabile `n` il numero di frasi da leggere specificato a linea di comando ed in `fortunefilename` il file da cui leggerle. Dopo aver installato (19-21) i gestori dei segnali per trattare l'uscita dal server, viene prima controllato (22) il numero di frasi richieste abbia senso (cioè sia maggiore di zero), le quali poi vengono lette (23) nel vettore in memoria con la stessa funzione `FortuneParse` usata anche per il server basato sulle *fifo*.

Una volta inizializzato il vettore di stringhe coi messaggi presi dal file delle *fortune* si procede (25) con la generazione di una chiave per identificare la coda di messaggi (si usa il nome del file dei sorgenti del server) con la quale poi si esegue (26) la creazione della stessa (si noti come si sia chiamata `msgget` con un valore opportuno per l'argomento `flag`), avendo cura di abortire il programma (27-29) in caso di errore.

Finita la fase di inizializzazione il server prima (32) chiama la funzione `daemon` per andare in background e poi esegue in permanenza il ciclo principale (33-40). Questo inizia (34) con il porsi in attesa di un messaggio di richiesta da parte di un client. Si noti infatti come `msgrcv` richieda un messaggio con `mtype` uguale a 1, questo è il valore usato per le richieste dato che corrisponde al *PID* di `init`, che non può essere un client. L'uso del flag `MSG_NOERROR` è solo per sicurezza, dato che i messaggi di richiesta sono di dimensione fissa (e contengono solo il *PID* del client).

Se non sono presenti messaggi di richiesta `msgrcv` si bloccherà, ritornando soltanto in corrispondenza dell'arrivo sulla coda di un messaggio di richiesta da parte di un client, in tal caso il ciclo prosegue (35) selezionando una frase a caso, copiandola (36) nella struttura `msgbuf_write` usata per la risposta e calcolandone (37) la dimensione.

Per poter permettere a ciascun client di ricevere solo la risposta indirizzata a lui il tipo del messaggio in uscita viene inizializzato (38) al valore del *PID* del client ricevuto nel messaggio di richiesta. L'ultimo passo del ciclo (39) è inviare sulla coda il messaggio di risposta. Si tenga conto che se la coda è piena anche questa funzione potrà bloccarsi fintanto che non venga liberato dello spazio.

Si noti che il programma può terminare solo grazie ad una interruzione da parte di un segnale; in tal caso verrà eseguito (45-48) il gestore `HandSIGTERM`, che semplicemente si limita a cancellare la coda (46) ed ad uscire (47).

In fig. 11.15 si è riportato un estratto il codice del programma client. Al solito il codice completo è con i sorgenti allegati, nel file `MQFortuneClient.c`. Come sempre si sono rimosse le parti relative alla gestione delle opzioni, ed in questo caso, anche la dichiarazione delle variabili, che, per la parte relative alle strutture usate per la comunicazione tramite le code, sono le stesse viste in fig. 11.14.

Il client in questo caso è molto semplice; la prima parte del programma (4-9) si occupa di accedere alla coda di messaggi, ed è identica a quanto visto per il server, solo che in questo caso `msgget` non viene chiamata con il flag di creazione in quanto la coda deve essere

```

1 int main(int argc, char *argv[])
2 {
3     ...
4     key = ftok("./MQFortuneServer.c", 1);
5     msgid = msgget(key, 0);
6     if (msgid < 0) {
7         perror("Cannot find message queue");
8         exit(1);
9     }
10    /* Main body: do request and write result */
11    msg_read.mtype = 1;                /* type for request is always 1 */
12    msg_read.pid = getpid();           /* use pid for communications */
13    size = sizeof(msg_read.pid);
14    msgsnd(msgid, &msg_read, size, 0); /* send request message */
15    msgrcv(msgid, &msg_write, MSGMAX, msg_read.pid, MSG_NOERROR);
16    printf("%s", msg_write.mtext);
17 }

```

Figura 11.15: Sezione principale del codice del client di *fortunes* basato sulle *message queue*.

preesistente. In caso di errore (ad esempio se il server non è stato avviato) il programma termina immediatamente.

Una volta acquisito l'identificatore della coda il client compone (12-13) il messaggio di richiesta in `msg_read`, usando 1 per il tipo ed inserendo il proprio *PID* come dato da passare al server. Calcolata (14) la dimensione, provvede (15) ad immettere la richiesta sulla coda.

A questo punto non resta che rileggere la risposta (16) dalla coda del server richiedendo a `msgrcv` di selezionare i messaggi di tipo corrispondente al valore del *PID* inviato nella richiesta. L'ultimo passo (17) prima di uscire è quello di stampare a video il messaggio ricevuto.

Proviamo allora il nostro nuovo sistema, al solito occorre definire `LD_LIBRARY_PATH` per accedere alla libreria `libgapil.so`, dopo di che, in maniera del tutto analoga a quanto fatto con il programma che usa le *fifo*, potremo far partire il server con:

```
[piccardi@gont sources]$ ./mqfortuned -n10
```

come nel caso precedente, avendo eseguito il server in background, il comando ritornerà immediatamente; potremo però verificare con `ps` che il programma è effettivamente in esecuzione, e che ha creato una coda di messaggi:

```
[piccardi@gont sources]$ ipcs
```

----- Shared Memory Segments -----						
key	shmid	owner	perms	bytes	nattch	status
----- Semaphore Arrays -----						
key	semid	owner	perms	nsems		
----- Message Queues -----						
key	msqid	owner	perms	used-bytes	messages	
0x0102dc6a	0	piccardi	666	0	0	

a questo punto potremo usare il client per ottenere le nostre frasi:

```
[piccardi@gont sources]$ ./mqfortune
Linux ext2fs has been stable for a long time, now it's time to break it
-- Linuxkongreß '95 in Berlin
```

```
[piccardi@gont sources]$ ./mqfortune
Let's call it an accidental feature.
--Larry Wall
```

con un risultato del tutto equivalente al precedente. Infine potremo chiudere il server inviando il segnale di terminazione con il comando `killall mqfortuned`, verificando che effettivamente la coda di messaggi venga rimossa.

Benché funzionante questa architettura risente dello stesso inconveniente visto anche nel caso del precedente server basato sulle *fifo*; se il client viene interrotto dopo l'invio del messaggio di richiesta e prima della lettura della risposta, quest'ultima resta nella coda (così come per le *fifo* si aveva il problema delle *fifo* che restavano nel filesystem). In questo caso però il problema è maggiore, sia perché è molto più facile esaurire la memoria dedicata ad una coda di messaggi che gli *inode* di un filesystem, sia perché, con il riutilizzo dei *PID* da parte dei processi, un client eseguito in un momento successivo potrebbe ricevere un messaggio non indirizzato a lui.

11.2.5 I semafori

I semafori non sono propriamente meccanismi di intercomunicazione come *pipe*, *fifo* e code di messaggi, poiché non consentono di scambiare dati fra processi, ma servono piuttosto come meccanismi di sincronizzazione o di protezione per le *sezioni critiche* del codice (si ricordi quanto detto in sez. 3.4.2). Un semaforo infatti non è altro che un contatore mantenuto nel kernel che determina se consentire o meno la prosecuzione dell'esecuzione di un programma. In questo modo si può controllare l'accesso ad una risorsa condivisa da più processi, associandovi un semaforo che assicuri che non possa essere usata da più di un processo alla volta.

Il concetto di semaforo è uno dei concetti base nella programmazione ed è assolutamente generico, così come del tutto generali sono modalità con cui lo si utilizza. Un processo che deve accedere ad una risorsa condivisa eseguirà un controllo del semaforo: se questo è positivo il suo valore sarà decrementato, indicando che si è consumato una unità della risorsa, ed il processo potrà proseguire nell'utilizzo di quest'ultima, provvedendo a rilasciarla, una volta completate le operazioni volute, reincrementando il semaforo.

Se al momento del controllo il valore del semaforo è nullo la risorsa viene considerata non disponibile, ed il processo si bloccherà fin quando chi la sta utilizzando non la rilascerà, incrementando il valore del semaforo. Non appena il semaforo diventa positivo, indicando che la risorsa è tornata disponibile, il processo bloccato in attesa riprenderà l'esecuzione, e potrà operare come nel caso precedente (decremento del semaforo, accesso alla risorsa, incremento del semaforo).

Per poter implementare questo tipo di logica le operazioni di controllo e decremento del contatore associato al semaforo devono essere atomiche, pertanto una realizzazione di un oggetto di questo tipo è necessariamente demandata al kernel. La forma più semplice di semaforo è quella del *semaforo binario*, o *mutex*, in cui un valore diverso da zero (normalmente 1) indica la libertà di accesso, e un valore nullo l'occupazione della risorsa. In generale però si possono usare semafori con valori interi, utilizzando il valore del contatore come indicatore del "numero di risorse" ancora disponibili.

Il sistema di intercomunicazione di *SysV-IPC* prevede anche una implementazione dei semafori, ma gli oggetti utilizzati sono tuttavia non semafori singoli, ma gruppi (più propriamente *insiemi*) di semafori detti "*semaphore set*". La funzione di sistema che permette di creare o ottenere l'identificatore di un insieme di semafori è `semget`, ed il suo prototipo è:

<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/sem.h> int semget(key_t key, int nsems, int flag)</pre>	
Restituisce l'identificatore di un insieme di semafori.	
La funzione ritorna l'identificatore (un intero positivo) in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:	
ENOSPC	si è superato il limite di sistema per il numero totale di semafori (<code>SEMMS</code>) o di insiemi (<code>SEMMNI</code>).
EINVAL	<code>nsems</code> è minore di zero o maggiore del limite sul numero di semafori di un insieme (<code>SEMMSL</code>), o se l'insieme già esiste, maggiore del numero di semafori che contiene.
ENOMEM	il sistema non ha abbastanza memoria per poter contenere le strutture per un nuovo insieme di semafori.
ed inoltre <code>EACCES</code> , <code>EEXIST</code> , <code>EIDRM</code> e <code>ENOENT</code> con lo stesso significato che hanno per <code>msgget</code> .	

La funzione è del tutto analoga a `msgget`, solo che in questo caso restituisce l'identificatore di un insieme di semafori, in particolare è identico l'uso degli argomenti `key` e `flag`, per cui non ripeteremo quanto detto al proposito in sez. 11.2.4. L'argomento `nsems` permette di specificare quanti semafori deve contenere l'insieme quando se ne richieda la creazione, e deve essere nullo quando si effettua una richiesta dell'identificatore di un insieme già esistente.

Purtroppo questa implementazione complica inutilmente lo schema elementare che abbiamo descritto, dato che non è possibile definire un singolo semaforo, ma se ne deve creare per forza un insieme. Ma questa in definitiva è solo una complicazione inutile dell'interfaccia, il problema è che i semafori forniti dal *SysV-IPC* soffrono di altri due difetti progettuali molto più gravi.

Il primo difetto è che non esiste una funzione che permetta di creare ed inizializzare un semaforo in un'unica chiamata; occorre prima creare l'insieme dei semafori con `semget` e poi inizializzarlo con `semctl`, si perde così ogni possibilità di eseguire l'operazione atomicamente. Eventuali accessi che possono avvenire fra la creazione e l'inizializzazione potranno avere effetti imprevisti.

Il secondo difetto deriva dalla caratteristica generale degli oggetti del *SysV-IPC* di essere risorse globali di sistema, che non vengono cancellate quando nessuno le usa più. In questo caso il problema è più grave perché ci si è a trovarsi a dover affrontare esplicitamente il caso in cui un processo termina per un qualche errore lasciando un semaforo occupato, che resterà tale fino al successivo riavvio del sistema. Come vedremo esistono delle modalità per evitare tutto ciò, ma diventa necessario indicare esplicitamente che si vuole il ripristino del semaforo all'uscita del processo, e la gestione diventa più complicata.

```
struct semid_ds
{
    struct ipc_perm sem_perm;      /* operation permission struct */
    time_t sem_otime;             /* last semop time */
    time_t sem_ctime;             /* last time changed by semctl */
    unsigned long int sem_nsems;  /* number of semaphores in set */
};
```

Figura 11.16: La struttura `semid_ds`, associata a ciascun insieme di semafori.

A ciascun insieme di semafori è associata una struttura `semid_ds`, riportata in fig. 11.16.¹¹

¹¹anche in questo caso in realtà il kernel usa una sua specifica struttura interna, ma i dati significativi sono

Come nel caso delle code di messaggi quando si crea un nuovo insieme di semafori con `semget` questa struttura viene inizializzata. In particolare il campo `sem_perm`, che esprime i permessi di accesso, viene inizializzato come illustrato in sez. 11.2.2 (si ricordi che in questo caso il permesso di scrittura è in realtà permesso di alterare il semaforo), per quanto riguarda gli altri campi invece:

- il campo `sem_nsems`, che esprime il numero di semafori nell'insieme, viene inizializzato al valore di `nsems`.
- il campo `sem_ctime`, che esprime il tempo di ultimo cambiamento dell'insieme, viene inizializzato al tempo corrente.
- il campo `sem_otime`, che esprime il tempo dell'ultima operazione effettuata, viene inizializzato a zero.

```

struct sem {
    short    sempid;          /* pid of last operation */
    ushort   semval;          /* current value */
    ushort   semncnt;         /* num procs awaiting increase in semval */
    ushort   semzcnt;         /* num procs awaiting semval = 0 */
};

```

Figura 11.17: La struttura `sem`, che contiene i dati di un singolo semaforo.

Ciascun semaforo dell'insieme è realizzato come una struttura di tipo `sem` che ne contiene i dati essenziali, la cui definizione è riportata in fig. 11.17.¹² Questa struttura non è accessibile direttamente dallo *user space*, ma i valori in essa specificati possono essere letti in maniera indiretta, attraverso l'uso delle opportune funzioni di controllo. I dati mantenuti nella struttura, ed elencati in fig. 11.17, indicano rispettivamente:

<code>semval</code>	il valore numerico del semaforo.
<code>sempid</code>	il <i>PID</i> dell'ultimo processo che ha eseguito una operazione sul semaforo.
<code>semncnt</code>	il numero di processi in attesa che esso venga incrementato.
<code>semzcnt</code>	il numero di processi in attesa che esso si annulli.

Come per le code di messaggi anche per gli insiemi di semafori esistono una serie di limiti, i cui valori sono associati ad altrettante costanti, che si sono riportate in tab. 11.2. Alcuni di questi limiti sono al solito accessibili e modificabili attraverso `sysctl` o scrivendo direttamente nel file `/proc/sys/kernel/sem`.

La funzione di sistema che permette di effettuare le varie operazioni di controllo sui semafori fra le quali, come accennato, è impropriamente compresa anche la loro inizializzazione, è `semctl`; il suo prototipo è:

sempre quelli citati.

¹²in realtà in fig 11.17 si è riportata la definizione originaria del kernel 1.0, che contiene la prima realizzazione del *SysV-IPC* in Linux; ormai questa struttura è ridotta ai soli due primi membri, e gli altri vengono calcolati dinamicamente, la si è usata solo a scopo di esempio, perché indica tutti i valori associati ad un semaforo, restituiti dalle funzioni di controllo, e citati dalle pagine di manuale.

Costante	Valore	Significato
SEMMNI	128	Numero massimo di insiemi di semafori.
SEMMSL	250	Numero massimo di semafori per insieme.
SEMNS	SEMMNI*SEMMSL	Numero massimo di semafori nel sistema.
SEMVMX	32767	Massimo valore per un semaforo.
SEMOPM	32	Massimo numero di operazioni per chiamata a semop.
SEMMNU	SEMMNS	Massimo numero di strutture di ripristino.
SEMUME	SEMOPM	Massimo numero di voci di ripristino.
SEMAEM	SEMVMX	Valore massimo per l'aggiustamento all'uscita.

Tabella 11.2: Valori delle costanti associate ai limiti degli insiemi di semafori, definite in `linux/sem.h`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd)
int semctl(int semid, int semnum, int cmd, union semun arg)
    Esegue le operazioni di controllo su un semaforo o un insieme di semafori.
```

La funzione ritorna in caso di successo un valore positivo quanto usata con tre argomenti ed un valore nullo quando usata con quattro e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES i permessi assegnati al semaforo non consentono l'operazione di lettura o scrittura richiesta e non si hanno i privilegi di amministratore.

EIDRM l'insieme di semafori è stato cancellato.

EPERM si è richiesto `IPC_SET` o `IPC_RMID` ma il processo non è né il creatore né il proprietario del semaforo e non ha i privilegi di amministratore.

ERANGE si è richiesto `SETALL` `SETVAL` ma il valore a cui si vuole impostare il semaforo è minore di zero o maggiore di `SEMVMX`.

ed inoltre `EFAULT` ed `EINVAL` nel loro significato generico.

La funzione può avere tre o quattro argomenti, a seconda dell'operazione specificata con `cmd`, ed opera o sull'intero insieme specificato da `semid` o sul singolo semaforo di un insieme, specificato da `semnum`.

```
union semun {
    int val; /* value for SETVAL */
    struct semid_ds *buf; /* buffer for IPC_STAT, IPC_SET */
    unsigned short *array; /* array for GETALL, SETALL */
    struct seminfo *__buf; /* buffer for IPC_INFO */
    /* (Linux specific) */
};
```

Figura 11.18: La definizione dei possibili valori di una union `semun`, usata come quarto argomento della funzione `semctl`.

Qualora la funzione operi con quattro argomenti `arg` è un argomento generico, che conterrà un dato diverso a seconda dell'azione richiesta; per unificare detto argomento esso deve essere passato come una unione `semun`, la cui definizione, con i possibili valori che può assumere, è riportata in fig. 11.18.

Nelle versioni più vecchie della *glibc* questa unione veniva definita in `sys/sem.h`, ma nelle versioni più recenti questo non avviene più in quanto lo standard POSIX.1-2001 richiede che

sia sempre definita a cura del chiamante. In questa seconda evenienza la *glibc* definisce però la macro `_SEM_SEMUN_UNDEFINED` che può essere usata per controllare la situazione.

Come già accennato sia il comportamento della funzione che il numero di argomenti con cui deve essere invocata dipendono dal valore dell'argomento `cmd`, che specifica l'azione da intraprendere. Per questo argomento i valori validi, quelli cioè che non causano un errore di `EINVAL`, sono i seguenti:

IPC_STAT	Legge i dati dell'insieme di semafori, copiandone i valori nella struttura <code>semid_ds</code> posta all'indirizzo specificato con <code>arg.buf</code> . Occorre avere il permesso di lettura. L'argomento <code>semnum</code> viene ignorato.
IPC_RMID	Rimuove l'insieme di semafori e le relative strutture dati, con effetto immediato. Tutti i processi che erano bloccati in attesa vengono svegliati, ritornando con un errore di <code>EIDRM</code> . L' <i>UID</i> effettivo del processo deve corrispondere o al creatore o al proprietario dell'insieme, o all'amministratore. L'argomento <code>semnum</code> viene ignorato.
IPC_SET	Permette di modificare i permessi ed il proprietario dell'insieme. I valori devono essere passati in una struttura <code>semid_ds</code> puntata da <code>arg.buf</code> di cui saranno usati soltanto i campi <code>sem_perm.uid</code> , <code>sem_perm.gid</code> e i nove bit meno significativi di <code>sem_perm.mode</code> . La funzione aggiorna anche il campo <code>sem_ctime</code> . L' <i>UID</i> effettivo del processo deve corrispondere o al creatore o al proprietario dell'insieme, o all'amministratore. L'argomento <code>semnum</code> viene ignorato.
GETALL	Restituisce il valore corrente di ciascun semaforo dell'insieme (corrispondente al campo <code>semval</code> di <code>sem</code>) nel vettore indicato da <code>arg.array</code> . Occorre avere il permesso di lettura. L'argomento <code>semnum</code> viene ignorato.
GETNCNT	Restituisce come valore di ritorno della funzione il numero di processi in attesa che il semaforo <code>semnum</code> dell'insieme <code>semid</code> venga incrementato (corrispondente al campo <code>semncnt</code> di <code>sem</code>). Va invocata con tre argomenti. Occorre avere il permesso di lettura.
GETPID	Restituisce come valore di ritorno della funzione il <i>PID</i> dell'ultimo processo che ha compiuto una operazione sul semaforo <code>semnum</code> dell'insieme <code>semid</code> (corrispondente al campo <code>sempid</code> di <code>sem</code>). Va invocata con tre argomenti. Occorre avere il permesso di lettura.
GETVAL	Restituisce come valore di ritorno della funzione il valore corrente del semaforo <code>semnum</code> dell'insieme <code>semid</code> (corrispondente al campo <code>semval</code> di <code>sem</code>). Va invocata con tre argomenti. Occorre avere il permesso di lettura.
GETZCNT	Restituisce come valore di ritorno della funzione il numero di processi in attesa che il valore del semaforo <code>semnum</code> dell'insieme <code>semid</code> diventi nullo (corrispondente al campo <code>semncnt</code> di <code>sem</code>). Va invocata con tre argomenti. Occorre avere il permesso di lettura.
SETALL	Inizializza il valore di tutti i semafori dell'insieme, aggiornando il campo <code>sem_ctime</code> di <code>semid_ds</code> . I valori devono essere passati nel vettore indicato da <code>arg.array</code> . Si devono avere i privilegi di scrittura. L'argomento <code>semnum</code> viene ignorato.
SETVAL	Inizializza il semaforo <code>semnum</code> al valore passato dall'argomento <code>arg.val</code> , aggiornando il campo <code>sem_ctime</code> di <code>semid_ds</code> . Si devono avere i privilegi di scrittura.

Come per `msgctl` esistono tre ulteriori valori, `IPC_INFO`, `SEM_STAT` e `SEM_INFO`, specifici di Linux e fuori da ogni standard, creati specificamente ad uso del comando `ipcs`. Dato che anche questi potranno essere modificati o rimossi, non devono essere utilizzati e pertanto non li tratteremo.

Quando si imposta il valore di un semaforo (sia che lo si faccia per tutto l'insieme con `SETALL`, che per un solo semaforo con `SETVAL`), i processi in attesa su di esso reagiscono di conseguenza al cambiamento di valore. Inoltre la coda delle operazioni di ripristino viene cancellata per tutti i semafori il cui valore viene modificato.

Operazione	Valore restituito
GETNCNT	Valore di <code>semncnt</code> .
GETPID	Valore di <code>sempid</code> .
GETVAL	Valore di <code>semval</code> .
GETZCNT	Valore di <code>semzcnt</code> .

Tabella 11.3: Valori di ritorno della funzione `semctl`.

Il valore di ritorno della funzione in caso di successo dipende dall'operazione richiesta; per tutte le operazioni che richiedono quattro argomenti esso è sempre nullo, per le altre operazioni, elencate in tab. 11.3 viene invece restituito il valore richiesto, corrispondente al campo della struttura `sem` indicato nella seconda colonna della tabella.

Le operazioni ordinarie sui semafori, come l'acquisizione o il rilascio degli stessi (in sostanza tutte quelle non comprese nell'uso di `semctl`) vengono effettuate con la funzione di sistema `semop`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops)
```

Esegue operazioni ordinarie su un semaforo o un insieme di semafori.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

E2BIG l'argomento `nsops` è maggiore del numero massimo di operazioni `SEMOPM`.

EACCES il processo non ha i permessi per eseguire l'operazione richiesta e non ha i privilegi di amministratore.

EAGAIN un'operazione comporterebbe il blocco del processo, ma si è specificato `IPC_NOWAIT` in `sem_flg`.

EFBIG il valore del campo `sem_num` è negativo o maggiore o uguale al numero di semafori dell'insieme.

EIDRM l'insieme di semafori è stato cancellato.

EINTR la funzione, bloccata in attesa dell'esecuzione dell'operazione, viene interrotta da un segnale.

ENOMEM si è richiesto un `SEM_UNDO` ma il sistema non ha le risorse per allocare la struttura di ripristino.

ERANGE per alcune operazioni il valore risultante del semaforo viene a superare il limite massimo `SEMVMX`.

ed inoltre `EFAULT` ed `EINVAL` nel loro significato generico.

La funzione permette di eseguire operazioni multiple sui singoli semafori di un insieme. La funzione richiede come primo argomento l'identificatore `semid` dell'insieme su cui si vuole operare, il numero di operazioni da effettuare viene specificato con l'argomento `nsop`, mentre il loro contenuto viene passato con un puntatore ad un vettore di strutture `sembuf` nell'argomento `sops`. Le operazioni richieste vengono effettivamente eseguite se e soltanto se è possibile

effettuarle tutte quante, ed in tal caso vengono eseguite nella sequenza passata nel vettore `sops`.

Con lo standard POSIX.1-2001 è stata introdotta una variante di `semop` che consente di specificare anche un tempo massimo di attesa. La nuova funzione di sistema, disponibile a partire dal kernel 2.4.22 e dalla *glibc* 2.3.3, ed utilizzabile solo dopo aver definito la macro `_GNU_SOURCE`, è `semtimedop`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semtimedop(int semid, struct sembuf *sops, unsigned nsops, struct timespec
*timeout)
```

Esegue operazioni ordinarie su un semaforo o un insieme di semafori.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN l'operazione comporterebbe il blocco del processo, ma si è specificato `IPC_NOWAIT` in `sem_flg` oppure si è atteso oltre quanto indicato da `timeout`.

e gli altri valori già visti per `semop`, con lo stesso significato.

Rispetto a `semop` la funzione consente di specificare un tempo massimo di attesa, indicato con una struttura `timespec` (vedi fig. 4.16), per le operazioni che verrebbero bloccate. Alla scadenza di detto tempo la funzione ritorna comunque con un errore di `EAGAIN` senza che nessuna delle operazioni richieste venga eseguita.

Si tenga presente che la precisione della temporizzazione è comunque limitata dalla risoluzione dell'orologio di sistema, per cui il tempo di attesa verrà arrotondato per eccesso. In caso si passi un valore `NULL` per `timeout` il comportamento di `semtimedop` è identico a quello di `semop`.

```
struct sembuf
{
    unsigned short int sem_num;    /* semaphore number */
    short int sem_op;             /* semaphore operation */
    short int sem_flg;            /* operation flag */
};
```

Figura 11.19: La struttura `sembuf`, usata per le operazioni sui semafori.

Come indicato il contenuto di ciascuna operazione deve essere specificato attraverso una struttura `sembuf` (la cui definizione è riportata in fig. 11.19) che il programma chiamante deve avere cura di allocare in un opportuno vettore. La struttura permette di indicare il semaforo su cui operare, il tipo di operazione, ed un flag di controllo.

Il campo `sem_num` serve per indicare a quale semaforo dell'insieme fa riferimento l'operazione. Si ricordi che i semafori sono numerati come gli elementi di un vettore, per cui il primo semaforo di un insieme corrisponde ad un valore nullo di `sem_num`.

Il campo `sem_flg` è un flag, mantenuto come maschera binaria, per il quale possono essere impostati i due valori `IPC_NOWAIT` e `SEM_UNDO`. Impostando `IPC_NOWAIT` si fa sì che in tutti quei casi in cui l'esecuzione di una operazione richiederebbe di porre il processo vada nello stato di *sleep*, invece di bloccarsi `semop` ritorni immediatamente (abortendo così le eventuali operazioni restanti) con un errore di `EAGAIN`. Impostando `SEM_UNDO` si richiede invece che l'operazione in questione venga registrata, in modo che il valore del semaforo possa essere ripristinato all'uscita del processo.

Infine `sem_op` è il campo che controlla qual'è l'operazione che viene eseguita e determina in generale il comportamento della chiamata a `semop`. I casi possibili per il valore di questo campo sono tre:

`sem_op > 0` In questo caso il valore viene aggiunto al valore corrente di `semval` per il semaforo indicato. Questa operazione non causa mai un blocco del processo, ed eventualmente `semop` ritorna immediatamente con un errore di `ERANGE` qualora si sia superato il limite `SEMVMX`. Se l'operazione ha successo si passa immediatamente alla successiva. Specificando `SEM_UNDO` si aggiorna il contatore per il ripristino del valore del semaforo. Al processo chiamante è richiesto il privilegio di alterazione (scrittura) sull'insieme di semafori.

`sem_op = 0` Nel caso `semval` sia zero l'operazione ha successo immediato, e o si passa alla successiva o `semop` ritorna con successo se questa era l'ultima. Se `semval` è diverso da zero il comportamento è controllato da `sem_flg`, se è stato impostato `IPC_NOWAIT` `semop` ritorna immediatamente abortendo tutte le operazioni con un errore di `EAGAIN`, altrimenti viene incrementato `semzcnt` di uno ed il processo viene bloccato fintanto che non si verifica una delle condizioni seguenti:

- `semval` diventa zero, nel qual caso `semzcnt` viene decrementato di uno, l'operazione ha successo e si passa alla successiva, oppure `semop` ritorna con successo se questa era l'ultima.
- l'insieme di semafori viene rimosso, nel qual caso `semop` ritorna abortendo tutte le operazioni con un errore di `EIDRM`.
- il processo chiamante riceve un segnale, nel qual caso `semzcnt` viene decrementato di uno e `semop` ritorna abortendo tutte le operazioni con un errore di `EINTR`.

Al processo chiamante è richiesto soltanto il privilegio di lettura dell'insieme dei semafori.

`sem_op < 0` Nel caso in cui `semval` è maggiore o uguale del valore assoluto di `sem_op` (se cioè la somma dei due valori resta positiva o nulla) i valori vengono sommati e l'operazione ha successo e si passa alla successiva, oppure `semop` ritorna con successo se questa era l'ultima. Qualora si sia impostato `SEM_UNDO` viene anche aggiornato il contatore per il ripristino del valore del semaforo. In caso contrario (quando cioè la somma darebbe luogo ad un valore di `semval` negativo) se si è impostato `IPC_NOWAIT` `semop` ritorna immediatamente abortendo tutte le operazioni con un errore di `EAGAIN`, altrimenti viene incrementato di uno `semncnt` ed il processo resta in stato di *sleep* fintanto che non si ha una delle condizioni seguenti:

- `semval` diventa maggiore o uguale del valore assoluto di `sem_op`, nel qual caso `semncnt` viene decrementato di uno, il valore di `sem_op` viene sommato a `semval`, e se era stato impostato `SEM_UNDO` viene aggiornato il contatore per il ripristino del valore del semaforo.
- l'insieme di semafori viene rimosso, nel qual caso `semop` ritorna abortendo tutte le operazioni con un errore di `EIDRM`.
- il processo chiamante riceve un segnale, nel qual caso `semncnt` viene decrementato di uno e `semop` ritorna abortendo tutte le operazioni con un errore di `EINTR`.

Al processo chiamante è richiesto il privilegio di alterazione (scrittura) sull'insieme di semafori.

Qualora si sia usato `semimedop` alle condizioni di errore precedenti si aggiunge anche quella di scadenza del tempo di attesa indicato con `timeout` che farà abortire la funzione, qualora resti bloccata troppo a lungo nell'esecuzione delle operazioni richieste, con un errore di `EAGAIN`.

In caso di successo (sia per `semop` che per `semimedop`) per ogni semaforo modificato verrà aggiornato il campo `sempid` al valore del `PID` del processo chiamante; inoltre verranno pure aggiornati al tempo corrente i campi `sem_otime` e `sem_ctime`.

Dato che, come già accennato in precedenza, in caso di uscita inaspettata i semafori possono restare occupati, abbiamo visto come `semop` (e `semimedop`) permetta di attivare un meccanismo di ripristino attraverso l'uso del flag `SEM_UNDO`. Il meccanismo è implementato tramite una apposita struttura `sem_undo`, associata ad ogni processo per ciascun semaforo che esso ha modificato; all'uscita i semafori modificati vengono ripristinati, e le strutture disallocate. Per mantenere coerente il comportamento queste strutture non vengono ereditate attraverso una `fork` (altrimenti si avrebbe un doppio ripristino), mentre passano inalterate nell'esecuzione di una `exec` (altrimenti non si avrebbe ripristino).

Tutto questo però ha un problema di fondo. Per capire di cosa si tratta occorre fare riferimento all'implementazione usata in Linux, che è riportata in maniera semplificata nello schema di fig. 11.20. Si è presa come riferimento l'architettura usata fino al kernel 2.2.x che è più semplice (ed illustrata in dettaglio in [?]). Nel kernel 2.4.x la struttura del *SysV-IPC* è stata modificata, ma le definizioni relative a queste strutture restano per compatibilità (in particolare con le vecchie versioni delle librerie del C, come le *libc5*).

Figura 11.20: Schema delle varie strutture di un insieme di semafori (`semid_ds`, `sem`, `sem_queue` e `sem_undo`).

Alla creazione di un nuovo insieme viene allocata una nuova struttura `semid_ds` ed il relativo vettore di strutture `sem`. Quando si richiede una operazione viene anzitutto verificato che tutte le operazioni possono avere successo; se una di esse comporta il blocco del processo il kernel crea una struttura `sem_queue` che viene aggiunta in fondo alla coda di attesa associata a ciascun insieme di semafori, che viene referenziata tramite i campi `sem_pending` e `sem_pending_last` di `semid_ds`. Nella struttura viene memorizzato il riferimento alle operazioni richieste (nel campo `sops`, che è un puntatore ad una struttura `sembuf`) e al processo corrente (nel campo `sleeper`) poi quest'ultimo viene messo stato di attesa e viene invocato lo *scheduler* per passare all'esecuzione di un altro processo.

Se invece tutte le operazioni possono avere successo vengono eseguite immediatamente, dopo di che il kernel esegue una scansione della coda di attesa (a partire da `sem_pending`) per verificare se qualcuna delle operazioni sospese in precedenza può essere eseguita, nel qual caso la struttura `sem_queue` viene rimossa e lo stato del processo associato all'operazione (`sleeper`) viene riportato a *running*; il tutto viene ripetuto fin quando non ci sono più operazioni eseguibili o si è svuotata la coda. Per gestire il meccanismo del ripristino tutte le volte che per un'operazione si è specificato il flag `SEM_UNDO` viene mantenuta per ciascun insieme di semafori una apposita struttura `sem_undo` che contiene (nel vettore puntato dal campo `semaadj`) un valore di aggiustamento per ogni semaforo cui viene sommato l'opposto del valore usato per l'operazione.

Queste strutture sono mantenute in due liste (rispettivamente attraverso i due campi `id_next` e `proc_next`) una associata all'insieme di cui fa parte il semaforo, che viene usata per invalidare le strutture se questo viene cancellato o per azzerarle se si è eseguita una

operazione con `semctl`, l'altra associata al processo che ha eseguito l'operazione, attraverso il campo `semundo` di `task_struct`, come mostrato in 11.20. Quando un processo termina, la lista ad esso associata viene scandita e le operazioni applicate al semaforo. Siccome un processo può accumulare delle richieste di ripristino per semafori differenti attraverso diverse chiamate a `semop`, si pone il problema di come eseguire il ripristino dei semafori all'uscita del processo, ed in particolare se questo può essere fatto atomicamente.

Il punto è cosa succede quando una delle operazioni previste per il ripristino non può essere eseguita immediatamente perché ad esempio il semaforo è occupato; in tal caso infatti, se si pone il processo in stato di *sleep* aspettando la disponibilità del semaforo (come faceva l'implementazione originaria) si perde l'atomicità dell'operazione. La scelta fatta dal kernel è pertanto quella di effettuare subito le operazioni che non prevedono un blocco del processo e di ignorare silenziosamente le altre; questo però comporta il fatto che il ripristino non è comunque garantito in tutte le occasioni.

Come esempio di uso dell'interfaccia dei semafori vediamo come implementare con essa dei semplici *mutex* (cioè semafori binari), tutto il codice in questione, contenuto nel file `Mutex.c` allegato ai sorgenti, è riportato in fig. 11.21. Utilizzeremo l'interfaccia per creare un insieme contenente un singolo semaforo, per il quale poi useremo un valore unitario per segnalare la disponibilità della risorsa, ed un valore nullo per segnalarne l'indisponibilità.

La prima funzione (2-15) è `MutexCreate` che data una chiave crea il semaforo usato per il mutex e lo inizializza, restituendone l'identificatore. Il primo passo (6) è chiamare `semget` con `IPC_CREATE` per creare il semaforo qualora non esista, assegnandogli i privilegi di lettura e scrittura per tutti. In caso di errore (7-9) si ritorna subito il risultato di `semget`, altrimenti (10) si inizializza il semaforo chiamando `semctl` con il comando `SETVAL`, utilizzando l'unione `semunion` dichiarata ed avvalorata in precedenza (4) ad 1 per significare che risorsa è libera. In caso di errore (11-13) si restituisce il valore di ritorno di `semctl`, altrimenti (14) si ritorna l'identificatore del semaforo.

La seconda funzione (17-20) è `MutexFind`, che, data una chiave, restituisce l'identificatore del semaforo ad essa associato. La comprensione del suo funzionamento è immediata in quanto essa è soltanto un *wrapper*¹³ di una chiamata a `semget` per cercare l'identificatore associato alla chiave, il valore di ritorno di quest'ultima viene passato all'indietro al chiamante.

La terza funzione (22-25) è `MutexRead` che, dato un identificatore, restituisce il valore del semaforo associato al mutex. Anche in questo caso la funzione è un *wrapper* per una chiamata a `semctl` con il comando `GETVAL`, che permette di restituire il valore del semaforo.

La quarta e la quinta funzione (36-44) sono `MutexLock`, e `MutexUnlock`, che permettono rispettivamente di bloccare e sbloccare il mutex. Entrambe fanno da wrapper per `semop`, utilizzando le due strutture `sem_lock` e `sem_unlock` definite in precedenza (27-34). Si noti come per queste ultime si sia fatto uso dell'opzione `SEM_UNDO` per evitare che il semaforo resti bloccato in caso di terminazione imprevista del processo.

L'ultima funzione (46-49) della serie, è `MutexRemove`, che rimuove il mutex. Anche in questo caso si ha un wrapper per una chiamata a `semctl` con il comando `IPC_RMID`, che permette di cancellare il semaforo; il valore di ritorno di quest'ultima viene passato all'indietro.

Chiamare `MutexLock` decrementa il valore del semaforo: se questo è libero (ha già valore 1) sarà bloccato (valore nullo), se è bloccato la chiamata a `semop` si bloccherà fintanto che la risorsa non venga rilasciata. Chiamando `MutexUnlock` il valore del semaforo sarà incrementato di uno, sbloccandolo qualora fosse bloccato.

¹³si chiama così una funzione usata per fare da *involucro* alla chiamata di un'altra, usata in genere per semplificare un'interfaccia (come in questo caso) o per utilizzare con la stessa funzione diversi substrati (librerie, ecc.) che possono fornire le stesse funzionalità.

```

1 /* Function MutexCreate: create a mutex/semaphore */
2 int MutexCreate(key_t ipc_key)
3 {
4     const union semun semunion={1}; /* semaphore union structure */
5     int sem_id, ret;
6     sem_id = semget(ipc_key, 1, IPC_CREAT|0666); /* get semaphore ID */
7     if (sem_id == -1) { /* if error return code */
8         return sem_id;
9     }
10    ret = semctl(sem_id, 0, SETVAL, semunion); /* init semaphore */
11    if (ret == -1) {
12        return ret;
13    }
14    return sem_id;
15 }
16 /* Function MutexFind: get the semaphore/mutex Id given the IPC key value */
17 int MutexFind(key_t ipc_key)
18 {
19     return semget(ipc_key,1,0);
20 }
21 /* Function MutexRead: read the current value of the mutex/semaphore */
22 int MutexRead(int sem_id)
23 {
24     return semctl(sem_id, 0, GETVAL);
25 }
26 /* Define sembuf structures to lock and unlock the semaphore */
27 struct sembuf sem_lock={ /* to lock semaphore */
28     0, /* semaphore number (only one so 0) */
29     -1, /* operation (-1 to use resource) */
30     SEM_UNDO}; /* flag (set for undo at exit) */
31 struct sembuf sem_unlock={ /* to unlock semaphore */
32     0, /* semaphore number (only one so 0) */
33     1, /* operation (1 to release resource) */
34     SEM_UNDO}; /* flag (in this case 0) */
35 /* Function MutexLock: to lock a mutex/semaphore */
36 int MutexLock(int sem_id)
37 {
38     return semop(sem_id, &sem_lock, 1);
39 }
40 /* Function MutexUnlock: to unlock a mutex/semaphore */
41 int MutexUnlock(int sem_id)
42 {
43     return semop(sem_id, &sem_unlock, 1);
44 }
45 /* Function MutexRemove: remove a mutex/semaphore */
46 int MutexRemove(int sem_id)
47 {
48     return semctl(sem_id, 0, IPC_RMID);
49 }

```

Figura 11.21: Il codice delle funzioni che permettono di creare o recuperare l'identificatore di un semaforo da utilizzare come *mutex*.

Si noti che occorre eseguire sempre prima `MutexLock` e poi `MutexUnlock`, perché se per un qualche errore si esegue più volte quest'ultima il valore del semaforo crescerebbe oltre 1, e `MutexLock` non avrebbe più l'effetto aspettato (bloccare la risorsa quando questa è considerata libera). Infine si tenga presente che usare `MutexRead` per controllare il valore dei mutex prima di proseguire in una operazione di sblocco non servirebbe comunque, dato che l'operazione non sarebbe atomica. Vedremo in sez. 11.3.3 come sia possibile ottenere un'interfaccia analoga a quella appena illustrata, senza incorrere in questi problemi, usando il *file locking*.

11.2.6 Memoria condivisa

Il terzo oggetto introdotto dal *SysV-IPC* è quello dei segmenti di memoria condivisa. La funzione di sistema che permette di ottenerne uno è `shmget`, ed il suo prototipo è:

<pre>#include <sys/types.h> #include <sys/ipc.h> #include <sys/shm.h> int shmget(key_t key, int size, int flag)</pre>	
	Ottiene o crea una memoria condivisa.
<p>La funzione ritorna l'identificatore (un intero positivo) in caso di successo e <code>-1</code> per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p>	
ENOSPC	si è superato il limite (<code>SHMMNI</code>) sul numero di segmenti di memoria nel sistema, o cercato di allocare un segmento le cui dimensioni fanno superare il limite di sistema (<code>SHMALL</code>) per la memoria ad essi riservata.
EINVAL	si è richiesta una dimensione per un nuovo segmento maggiore di <code>SHMMAX</code> o minore di <code>SHMMIN</code> , o se il segmento già esiste <code>size</code> è maggiore delle sue dimensioni.
ENOMEM	il sistema non ha abbastanza memoria per poter contenere le strutture per un nuovo segmento di memoria condivisa.
ENOMEM	si è specificato <code>IPC_HUGETLB</code> ma non si hanno i privilegi di amministratore.
ed inoltre <code>EACCES</code> , <code>ENOENT</code> , <code>EEXIST</code> , <code>EIDRM</code> , con lo stesso significato che hanno per <code>msgget</code> .	

La funzione, come `semget`, è analoga a `msgget`, ed identico è l'uso degli argomenti `key` e `flag` per cui non ripeteremo quanto detto al proposito in sez. 11.2.4. A partire dal kernel 2.6 però sono stati introdotti degli ulteriori bit di controllo per l'argomento `flag`, specifici di `shmget`, attinenti alle modalità di gestione del segmento di memoria condivisa in relazione al sistema della memoria virtuale.

Il primo dei due flag è `SHM_HUGETLB` che consente di richiedere la creazione del segmento usando una *huge page*, le pagine di memoria di grandi dimensioni introdotte con il kernel 2.6 per ottimizzare le prestazioni nei sistemi più recenti che hanno grandi quantità di memoria. L'operazione è privilegiata e richiede che il processo abbia la *capability* `CAP_IPC_LOCK`. Questa funzionalità è specifica di Linux e non è portabile.

Il secondo flag aggiuntivo, introdotto a partire dal kernel 2.6.15, è `SHM_NORESERVE`, ed ha lo stesso scopo del flag `MAP_NORESERVE` di `mmap` (vedi sez. 10.4.1): non vengono riservate delle pagine di swap ad uso del meccanismo del *copy on write* per mantenere le modifiche fatte sul segmento. Questo significa che caso di scrittura sul segmento quando non c'è più memoria disponibile, si avrà l'emissione di un `SIGSEGV`.

Infine l'argomento `size` specifica la dimensione del segmento di memoria condivisa; il valore deve essere specificato in byte, ma verrà comunque arrotondato al multiplo superiore di `PAGE_SIZE`. Il valore deve essere specificato quando si crea un nuovo segmento di memoria con `IPC_CREAT` o `IPC_PRIVATE`, se invece si accede ad un segmento di memoria condivisa esistente non può essere maggiore del valore con cui esso è stato creato.

La memoria condivisa è la forma più veloce di comunicazione fra due processi, in quanto permette agli stessi di vedere nel loro spazio di indirizzi una stessa sezione di memoria.

Pertanto non è necessaria nessuna operazione di copia per trasmettere i dati da un processo all'altro, in quanto ciascuno può accedervi direttamente con le normali operazioni di lettura e scrittura dei dati in memoria.

Ovviamente tutto questo ha un prezzo, ed il problema fondamentale della memoria condivisa è la sincronizzazione degli accessi. È evidente infatti che se un processo deve scambiare dei dati con un altro, si deve essere sicuri che quest'ultimo non acceda al segmento di memoria condivisa prima che il primo non abbia completato le operazioni di scrittura, inoltre nel corso di una lettura si deve essere sicuri che i dati restano coerenti e non vengono sovrascritti da un accesso in scrittura sullo stesso segmento da parte di un altro processo. Per questo in genere la memoria condivisa viene sempre utilizzata in abbinamento ad un meccanismo di sincronizzazione, il che, di norma, significa insieme a dei semafori.

```

struct shmid_ds {
    struct ipc_perm shm_perm; /* operation perms */
    int shm_segsz; /* size of segment (bytes) */
    time_t shm_atime; /* last attach time */
    time_t shm_dtime; /* last detach time */
    time_t shm_ctime; /* last change time */
    unsigned short shm_cpid; /* pid of creator */
    unsigned short shm_lpid; /* pid of last operator */
    short shm_nattch; /* no. of current attaches */
};

```

Figura 11.22: La struttura `shmid_ds`, associata a ciascun segmento di memoria condivisa.

A ciascun segmento di memoria condivisa è associata una struttura `shmid_ds`, riportata in fig. 11.22. Come nel caso delle code di messaggi quando si crea un nuovo segmento di memoria condivisa con `shmget` questa struttura viene inizializzata, in particolare il campo `shm_perm` viene inizializzato come illustrato in sez. 11.2.2, e valgono le considerazioni ivi fatte relativamente ai permessi di accesso; per quanto riguarda gli altri campi invece:

- il campo `shm_segsz`, che esprime la dimensione del segmento, viene inizializzato al valore di `size`.
- il campo `shm_ctime`, che esprime il tempo di creazione del segmento, viene inizializzato al tempo corrente.
- i campi `shm_atime` e `shm_dtime`, che esprimono rispettivamente il tempo dell'ultima volta che il segmento è stato agganciato o sganciato da un processo, vengono inizializzati a zero.
- il campo `shm_lpid`, che esprime il *PID* del processo che ha eseguito l'ultima operazione, viene inizializzato a zero.
- il campo `shm_cpid`, che esprime il *PID* del processo che ha creato il segmento, viene inizializzato al *PID* del processo chiamante.
- il campo `shm_nattac`, che esprime il numero di processi agganciati al segmento viene inizializzato a zero.

Come per le code di messaggi e gli insiemi di semafori, anche per i segmenti di memoria condivisa esistono una serie di limiti imposti dal sistema. Alcuni di questi limiti sono al solito accessibili e modificabili attraverso `sysctl` o scrivendo direttamente nei rispettivi file di `/proc/sys/kernel/`.

In tab. 11.4 si sono riportate le costanti simboliche associate a ciascuno di essi, il loro significato, i valori preimpostati, e, quando presente, il file in `/proc/sys/kernel/` che permettono di cambiarne il valore.

Costante	Valore	File in proc	Significato
SHMALL	0x200000	shmall	Numero massimo di pagine che possono essere usate per i segmenti di memoria condivisa.
SHMMAX	0x2000000	shmmx	Dimensione massima di un segmento di memoria condivisa.
SHMMNI	4096	shmmni	Numero massimo di segmenti di memoria condivisa presenti nel kernel.
SHMMIN	1	—	Dimensione minima di un segmento di memoria condivisa.
SHMLBA	PAGE_SIZE	—	Limite inferiore per le dimensioni minime di un segmento (deve essere allineato alle dimensioni di una pagina di memoria).
SHMSEG	—	—	Numero massimo di segmenti di memoria condivisa per ciascun processo (l'implementazione non prevede l'esistenza di questo limite).

Tabella 11.4: Valori delle costanti associate ai limiti dei segmenti di memoria condivisa, insieme al relativo file in `/proc/sys/kernel/` ed al valore preimpostato presente nel sistema.

Al solito la funzione di sistema che permette di effettuare le operazioni di controllo su un segmento di memoria condivisa è `shmctl`; il suo prototipo è:

<pre>#include <sys/ipc.h> #include <sys/shm.h> int shmctl(int shmid, int cmd, struct shmid_ds *buf)</pre> <p>Esegue le operazioni di controllo su un segmento di memoria condivisa.</p>	
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p>	
EACCES	si è richiesto IPC_STAT ma i permessi non consentono l'accesso in lettura al segmento.
EFAULT	l'indirizzo specificato con <code>buf</code> non è valido.
EIDRM	l'argomento <code>shmid</code> fa riferimento ad un segmento che è stato cancellato.
EINVAL	o <code>shmid</code> non è un identificatore valido o <code>cmd</code> non è un comando valido.
ENOMEM	si è richiesto un <i>memory lock</i> di dimensioni superiori al massimo consentito.
EOVERFLOW	si è tentato il comando IPC_STAT ma il valore del <i>GID</i> o dell' <i>UID</i> è troppo grande per essere memorizzato nella struttura puntata da <code>buf</code> .
EPERM	si è specificato un comando con IPC_SET o IPC_RMID senza i permessi necessari.

Il comando specificato attraverso l'argomento `cmd` determina i diversi effetti della funzione. Nello standard POSIX.1-2001 i valori che esso può assumere, ed il corrispondente comportamento della funzione, sono i seguenti:

IPC_STAT	Legge le informazioni riguardo il segmento di memoria condivisa nella struttura <code>shmid_ds</code> puntata da <code>buf</code> . Occorre che il processo chiamante abbia il permesso di lettura sulla segmento.
IPC_RMID	Marca il segmento di memoria condivisa per la rimozione, questo verrà cancellato effettivamente solo quando l'ultimo processo ad esso agganciato si sarà staccato. Questo comando può essere eseguito solo da un processo con <i>UID</i> effettivo corrispondente o al creatore del segmento, o al proprietario del segmento, o all'amministratore.

IPC_SET Permette di modificare i permessi ed il proprietario del segmento. Per modificare i valori di `shm_perm.mode`, `shm_perm.uid` e `shm_perm.gid` occorre essere il proprietario o il creatore del segmento, oppure l'amministratore. Compiuta l'operazione aggiorna anche il valore del campo `shm_ctime`.

Oltre ai precedenti su Linux sono definiti anche degli ulteriori comandi, che consentono di estendere le funzionalità, ovviamente non devono essere usati se si ha a cuore la portabilità. Questi comandi aggiuntivi sono:

SHM_LOCK Abilita il *memory locking* sul segmento di memoria condivisa, impedendo che la memoria usata per il segmento venga salvata su disco dal meccanismo della memoria virtuale. Come illustrato in sez. 2.2.4 fino al kernel 2.6.9 solo l'amministratore poteva utilizzare questa capacità,¹⁴ a partire dal kernel 2.6.10 anche gli utenti normali possono farlo fino al limite massimo determinato da `RLIMIT_MEMLOCK` (vedi sez. 6.3.2).

SHM_UNLOCK Disabilita il *memory locking* sul segmento di memoria condivisa. Fino al kernel 2.6.9 solo l'amministratore poteva utilizzare questo comando in corrispondenza di un segmento da lui bloccato.

A questi due, come per `msgctl` e `semctl`, si aggiungono tre ulteriori valori, `IPC_INFO`, `SHM_STAT` e `SHM_INFO`, introdotti ad uso del programma `ipcs` per ottenere le informazioni generali relative alle risorse usate dai segmenti di memoria condivisa. Dato che potranno essere modificati o rimossi in favore dell'uso di `/proc`, non devono essere usati e non li tratteremo.

L'argomento `buf` viene utilizzato solo con i comandi `IPC_STAT` e `IPC_SET` nel qual caso esso dovrà puntare ad una struttura `shmid_ds` precedentemente allocata, in cui nel primo caso saranno scritti i dati del segmento di memoria restituiti dalla funzione e da cui, nel secondo caso, verranno letti i dati da impostare sul segmento.

Una volta che lo si è creato, per utilizzare un segmento di memoria condivisa l'interfaccia prevede due funzioni, `shmat` e `shmdt`. La prima di queste serve ad agganciare un segmento al processo chiamante, in modo che quest'ultimo possa inserirlo nel suo spazio di indirizzi per potervi accedere; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg)
    Aggancia un segmento di memoria condivisa al processo chiamante.
```

La funzione ritorna l'indirizzo del segmento in caso di successo e `-1` (in un cast a `void *`) per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES il processo non ha i privilegi per accedere al segmento nella modalità richiesta.

EINVAL si è specificato un identificatore invalido per `shmid`, o un indirizzo non allineato sul confine di una pagina per `shmaddr` o il valore `NULL` indicando `SHM_REMAP`.

ed inoltre `ENOMEM` nel suo significato generico.

La funzione inserisce un segmento di memoria condivisa all'interno dello spazio di indirizzi del processo, in modo che questo possa accedervi direttamente, la situazione dopo l'esecuzione di `shmat` è illustrata in fig. 11.23 (per la comprensione del resto dello schema si ricordi quanto illustrato al proposito in sez. 2.2.2). In particolare l'indirizzo finale del segmento dati (quello impostato da `brk`, vedi sez. 2.2.3) non viene influenzato. Si tenga presente infine che la funzione ha successo anche se il segmento è stato marcato per la cancellazione.

¹⁴che richiedeva la *capability* `CAP_IPC_LOCK`.

Figura 11.23: Disposizione dei segmenti di memoria di un processo quando si è agganciato un segmento di memoria condivisa.

L'argomento `shmaddr` specifica a quale indirizzo¹⁵ deve essere associato il segmento, se il valore specificato è `NULL` è il sistema a scegliere opportunamente un'area di memoria libera (questo è il modo più portabile e sicuro di usare la funzione). Altrimenti il kernel aggancia il segmento all'indirizzo specificato da `shmaddr`; questo però può avvenire solo se l'indirizzo coincide con il limite di una pagina, cioè se è un multiplo esatto del parametro di sistema `SHMLBA`, che in Linux è sempre uguale `PAGE_SIZE`.

Si tenga presente però che quando si usa `NULL` come valore di `shmaddr`, l'indirizzo restituito da `shmat` può cambiare da processo a processo; pertanto se nell'area di memoria condivisa si salvano anche degli indirizzi, si deve avere cura di usare valori relativi (in genere riferiti all'indirizzo di partenza del segmento).

L'argomento `shmflg` permette di cambiare il comportamento della funzione; esso va specificato come maschera binaria, i bit utilizzati al momento sono tre e sono identificati dalle costanti `SHM_RND`, `SHM_RDONLY` e `SHM_REMAP` che vanno combinate con un OR aritmetico.

Specificando `SHM_RND` si evita che `shmat` ritorni un errore quando `shmaddr` non è allineato ai confini di una pagina. Si può quindi usare un valore qualunque per `shmaddr`, e il segmento verrà comunque agganciato, ma al più vicino multiplo di `SHMLBA`; il nome della costante sta infatti per *rounded*, e serve per specificare un indirizzo come arrotondamento.

L'uso di `SHM_RDONLY` permette di agganciare il segmento in sola lettura (si ricordi che anche le pagine di memoria hanno dei permessi), in tal caso un tentativo di scrivere sul segmento comporterà una violazione di accesso con l'emissione di un segnale di `SIGSEGV`. Il comportamento usuale di `shmat` è quello di agganciare il segmento con l'accesso in lettura e scrittura (ed il processo deve aver questi permessi in `shm_perm`), non è prevista la possibilità di agganciare un segmento in sola scrittura.

Infine `SHM_REMAP` è una estensione specifica di Linux (quindi non portabile) che indica che la mappatura del segmento deve rimpiazzare ogni precedente mappatura esistente nell'intervallo iniziante all'indirizzo `shmaddr` e di dimensione pari alla lunghezza del segmento. In condizioni normali questo tipo di richiesta fallirebbe con un errore di `EINVAL`. Ovviamente usando `SHM_REMAP` l'argomento `shmaddr` non può essere nullo.

In caso di successo la funzione `shmat` aggiorna anche i seguenti campi della struttura `shmid_ds`:

- il tempo `shm_atime` dell'ultima operazione di aggancio viene impostato al tempo corrente.
- il `PID` `shm_lpid` dell'ultimo processo che ha operato sul segmento viene impostato a quello del processo corrente.
- il numero `shm_nattch` di processi agganciati al segmento viene aumentato di uno.

Come accennato in sez. 3.1.3 un segmento di memoria condivisa agganciato ad un processo viene ereditato da un figlio attraverso una `fork`, dato che quest'ultimo riceve una copia dello spazio degli indirizzi del padre. Invece, dato che attraverso una `exec` viene eseguito un diverso programma con uno spazio di indirizzi completamente diverso, tutti i segmenti agganciati al processo originario vengono automaticamente sagganciati. Lo stesso avviene all'uscita del processo attraverso una `exit`.

¹⁵lo standard SVI¹D prevede che l'argomento `shmaddr` sia di tipo `char *`, così come il valore di ritorno della funzione; in Linux è stato così con la *libc4* e la *libc5*, con il passaggio alla *glibc* il tipo di `shmaddr` è divenuto un `const void *` e quello del valore di ritorno un `void *` seguendo POSIX.1-2001.

Una volta che un segmento di memoria condivisa non serve più, si può sganciarlo esplicitamente dal processo usando l'altra funzione dell'interfaccia, `shmdt`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr)
```

Sgancia dal processo un segmento di memoria condivisa.

La funzione ritorna 0 in caso di successo e -1 per un errore, la funzione fallisce solo quando non c'è un segmento agganciato all'indirizzo `shmaddr`, con `errno` che assume il valore `EINVAL`.

La funzione sgancia dallo spazio degli indirizzi del processo un segmento di memoria condivisa; questo viene identificato con l'indirizzo `shmaddr` restituito dalla precedente chiamata a `shmat` con il quale era stato agganciato al processo.

In caso di successo la funzione aggiorna anche i seguenti campi di `shmid_ds`:

- il tempo `shm_dtime` dell'ultima operazione di sganciamento viene impostato al tempo corrente.
- il `PID shm_lpid` dell'ultimo processo che ha operato sul segmento viene impostato a quello del processo corrente.
- il numero `shm_nattch` di processi agganciati al segmento viene decrementato di uno.

inoltre la regione di indirizzi usata per il segmento di memoria condivisa viene tolta dallo spazio di indirizzi del processo.

Come esempio di uso di queste funzioni vediamo come implementare una serie di funzioni di libreria che ne semplifichino l'uso, automatizzando le operazioni più comuni; il codice, contenuto nel file `SharedMem.c`, è riportato in fig. 11.24.

La prima funzione (1-16) è `ShmCreate` che, data una chiave, crea il segmento di memoria condivisa restituendo il puntatore allo stesso. La funzione comincia (6) con il chiamare `shmget`, usando il flag `IPC_CREATE` per creare il segmento qualora non esista, ed assegnandogli i privilegi specificati dall'argomento `perm` e la dimensione specificata dall'argomento `shm_size`. In caso di errore (7-9) si ritorna immediatamente un puntatore nullo, altrimenti (10) si prosegue agganciando il segmento di memoria condivisa al processo con `shmat`. In caso di errore (11-13) si restituisce di nuovo un puntatore nullo, infine (14) si inizializza con `memset` il contenuto del segmento al valore costante specificato dall'argomento `fill`, e poi si ritorna il puntatore al segmento stesso.

La seconda funzione (17-31) è `ShmFind`, che, data una chiave, restituisce l'indirizzo del segmento ad essa associato. Anzitutto (22) si richiede l'identificatore del segmento con `shmget`, ritornando (23-25) un puntatore nullo in caso di errore. Poi si prosegue (26) agganciando il segmento al processo con `shmat`, restituendo (27-29) di nuovo un puntatore nullo in caso di errore, se invece non ci sono errori si restituisce il puntatore ottenuto da `shmat`.

La terza funzione (32-51) è `ShmRemove` che, data la chiave ed il puntatore associati al segmento di memoria condivisa, prima lo sgancia dal processo e poi lo rimuove. Il primo passo (37) è la chiamata a `shmdt` per sganciare il segmento, restituendo (38-39) un valore -1 in caso di errore. Il passo successivo (41) è utilizzare `shmget` per ottenere l'identificatore associato al segmento data la chiave `key`. Al solito si restituisce un valore di -1 (42-45) in caso di errore, mentre se tutto va bene si conclude restituendo un valore nullo.

Benché la memoria condivisa costituisca il meccanismo di intercomunicazione fra processi più veloce, essa non è sempre il più appropriato, dato che, come abbiamo visto, si avrà comunque la necessità di una sincronizzazione degli accessi. Per questo motivo, quando la comunicazione fra processi è sequenziale, altri meccanismi come le *pipe*, le *fifo* o i *socket*, che non necessitano di sincronizzazione esplicita, sono da preferire. Essa diventa l'unico meccanismo

```

1  /* Function ShmCreate  Create a SysV shared memory segment */
2  void * ShmCreate(key_t ipc_key, int shm_size, int perm, int fill)
3  {
4      void * shm_ptr;
5      int shm_id;      /* ID of the IPC shared memory segment */
6      shm_id = shmget(ipc_key, shm_size, IPC_CREAT|perm); /* get shm ID */
7      if (shm_id < 0) {
8          return NULL;
9      }
10     shm_ptr = shmat(shm_id, NULL, 0);      /* map it into memory */
11     if (shm_ptr < 0) {
12         return NULL;
13     }
14     memset((void *)shm_ptr, fill, shm_size); /* fill segment */
15     return shm_ptr;
16 }
17 /* Function ShmFind: Find a SysV shared memory segment */
18 void * ShmFind(key_t ipc_key, int shm_size)
19 {
20     void * shm_ptr;
21     int shm_id;      /* ID of the SysV shared memory segment */
22     shm_id = shmget(ipc_key, shm_size, 0); /* find shared memory ID */
23     if (shm_id < 0) {
24         return NULL;
25     }
26     shm_ptr = shmat(shm_id, NULL, 0);      /* map it into memory */
27     if (shm_ptr < 0) {
28         return NULL;
29     }
30     return shm_ptr;
31 }
32 /* Function ShmRemove: Schedule removal for a SysV shared memory segment */
33 int ShmRemove(key_t ipc_key, void * shm_ptr)
34 {
35     int shm_id;      /* ID of the SysV shared memory segment */
36     /* first detach segment */
37     if (shmdt(shm_ptr) < 0) {
38         return -1;
39     }
40     /* schedule segment removal */
41     shm_id = shmget(ipc_key, 0, 0);      /* find shared memory ID */
42     if (shm_id < 0) {
43         if (errno == EIDRM) return 0;
44         return -1;
45     }
46     if (shmctl(shm_id, IPC_RMID, NULL) < 0) { /* ask for removal */
47         if (errno == EIDRM) return 0;
48         return -1;
49     }
50     return 0;
51 }

```

Figura 11.24: Il codice delle funzioni che permettono di creare, trovare e rimuovere un segmento di memoria condivisa.

possibile quando la comunicazione non è sequenziale¹⁶ o quando non può avvenire secondo una modalità predefinita.

Un esempio classico di uso della memoria condivisa è quello del “*monitor*”, in cui viene per scambiare informazioni fra un processo server, che vi scrive dei dati di interesse generale che ha ottenuto, e i processi client interessati agli stessi dati che così possono leggerli in maniera completamente asincrona. Con questo schema di funzionamento da una parte si evita che ciascun processo client debba compiere l'operazione, potenzialmente onerosa, di ricavare e trattare i dati, e dall'altra si evita al processo server di dover gestire l'invio a tutti i client di tutti i dati (non potendo il server sapere quali di essi servono effettivamente al singolo client).

Nel nostro caso implementeremo un “*monitor*” di una *directory*: un processo si incaricherà di tenere sotto controllo alcuni parametri relativi ad una *directory* (il numero dei file contenuti, la dimensione totale, quante *directory*, link simbolici, file normali, ecc.) che saranno salvati in un segmento di memoria condivisa cui altri processi potranno accedere per ricavare la parte di informazione che interessa.

In fig. 11.25 si è riportata la sezione principale del corpo del programma server, insieme alle definizioni delle altre funzioni usate nel programma e delle variabili globali, omettendo tutto quello che riguarda la gestione delle opzioni e la stampa delle istruzioni di uso a video; al solito il codice completo si trova con i sorgenti allegati nel file `DirMonitor.c`.

Il programma usa delle variabili globali (2-14) per mantenere i valori relativi agli oggetti usati per la comunicazione inter-processo; si è definita inoltre una apposita struttura `DirProp` che contiene i dati relativi alle proprietà che si vogliono mantenere nella memoria condivisa, per l'accesso da parte dei client.

Il programma, dopo la sezione, omessa, relativa alla gestione delle opzioni da riga di comando (che si limitano alla eventuale stampa di un messaggio di aiuto a video ed all'impostazione della durata dell'intervallo con cui viene ripetuto il calcolo delle proprietà della *directory*) controlla (20-23) che sia stato specificato l'argomento necessario contenente il nome della *directory* da tenere sotto controllo, senza il quale esce immediatamente con un messaggio di errore.

Poi, per verificare che l'argomento specifichi effettivamente una *directory*, si esegue (24-26) su di esso una `chdir`, uscendo immediatamente in caso di errore. Questa funzione serve anche per impostare la *directory* di lavoro del programma nella *directory* da tenere sotto controllo, in vista del successivo uso della funzione `daemon`. Si noti come si è potuta fare questa scelta, nonostante le indicazioni illustrate in sez. 8.1.5, per il particolare scopo del programma, che necessita comunque di restare all'interno di una *directory*.

Infine (27-29) si installano i gestori per i vari segnali di terminazione che, avendo a che fare con un programma che deve essere eseguito come server, sono il solo strumento disponibile per concluderne l'esecuzione.

Il passo successivo (30-39) è quello di creare gli oggetti di intercomunicazione necessari. Si inizia costruendo (30) la chiave da usare come riferimento con il nome del programma,¹⁷ dopo di che si richiede (31) la creazione di un segmento di memoria condivisa con usando la funzione `ShmCreate` illustrata in precedenza (una pagina di memoria è sufficiente per i dati che useremo), uscendo (32-35) qualora la creazione ed il successivo agganciamento al processo non abbia successo. Con l'indirizzo `shmptr` così ottenuto potremo poi accedere alla memoria condivisa, che, per come abbiamo lo abbiamo definito, sarà vista nella forma data da `DirProp`. Infine (36-39) utilizzando sempre la stessa chiave, si crea, tramite le funzioni di interfaccia già

¹⁶come accennato in sez. 11.2.4 per la comunicazione non sequenziale si possono usare le code di messaggi, attraverso l'uso del campo `mtype`, ma solo se quest'ultima può essere effettuata in forma di messaggio.

¹⁷si è usato un riferimento relativo alla home dell'utente, supposto che i sorgenti di GaPiL siano stati installati direttamente in essa; qualora si effettui una installazione diversa si dovrà correggere il programma.

```

1 /* global variables for shared memory segment */
2 struct DirProp {
3     int tot_size;
4     int tot_files;
5     int tot_regular;
6     int tot_fifo;
7     int tot_link;
8     int tot_dir;
9     int tot_block;
10    int tot_char;
11    int tot_sock;
12 } *shmptr;
13 key_t key;
14 int mutex;
15 /* main body */
16 int main(int argc, char *argv[])
17 {
18     int i, pause = 10;
19     ...
20     if ((argc - optind) != 1) { /* There must be remaing parameters */
21         printf("Wrong_number_of_arguments_%d\n", argc - optind);
22         usage();
23     }
24     if (chdir(argv[1])) { /* chdir to be sure dir exist */
25         perror("Cannot_find_directory_to_monitor");
26     }
27     Signal(SIGTERM, HandSIGTERM); /* set handlers for termination */
28     Signal(SIGINT, HandSIGTERM);
29     Signal(SIGQUIT, HandSIGTERM);
30     key = ftok("~/gatil/sources/DirMonitor.c", 1); /* define a key */
31     shmptr = ShmCreate(key, 4096, 0666, 0); /* get a shared memory segment */
32     if (!shmptr) {
33         perror("Cannot_create_shared_memory");
34         exit(1);
35     }
36     if ((mutex = MutexCreate(key)) == -1) { /* get a Mutex */
37         perror("Cannot_create_mutex");
38         exit(1);
39     }
40     /* main loop, monitor directory properties each 10 sec */
41     daemon(1, 0); /* demonize process, staying in monitored dir */
42     while (1) {
43         MutexLock(mutex); /* lock shared memory */
44         memset(shmptr, 0, sizeof(struct DirProp)); /* erase previous data */
45         dir_scan(argv[1], ComputeValues); /* execute scan */
46         MutexUnlock(mutex); /* unlock shared memory */
47         sleep(pause); /* sleep until next watch */
48     }
49 }

```

Figura 11.25: Codice della funzione principale del programma DirMonitor.c.

descritte in sez. 11.2.5, anche un mutex, che utilizzeremo per regolare l'accesso alla memoria condivisa.

Completata l'inizializzazione e la creazione degli oggetti di intercomunicazione il programma entra nel ciclo principale (40-49) dove vengono eseguite indefinitamente le attività di monitoraggio. Il primo passo (41) è eseguire `daemon` per proseguire con l'esecuzione in background come si conviene ad un programma demone; si noti che si è mantenuta, usando un valore non nullo del primo argomento, la directory di lavoro corrente. Una volta che il programma è andato in background l'esecuzione prosegue all'interno di un ciclo infinito (42-48).

Si inizia (43) bloccando il mutex con `MutexLock` per poter accedere alla memoria condivisa (la funzione si bloccherà automaticamente se qualche client sta leggendo), poi (44) si cancellano i valori precedentemente immagazzinati nella memoria condivisa con `memset`, e si esegue (45) un nuovo calcolo degli stessi utilizzando la funzione `dir_scan`; infine (46) si sblocca il mutex con `MutexUnlock`, e si attende (47) per il periodo di tempo specificato a riga di comando con l'opzione `-p` usando una `sleep`.

Si noti come per il calcolo dei valori da mantenere nella memoria condivisa si sia usata ancora una volta la funzione `dir_scan`, già utilizzata (e descritta in dettaglio) in sez. 4.2.3, che ci permette di effettuare la scansione delle voci della directory, chiamando per ciascuna di esse la funzione `ComputeValues`, che esegue tutti i calcoli necessari.

```

1  /* Routine to compute directory properties inside dir_scan */
2  int ComputeValues(struct dirent * direntry)
3  {
4      struct stat data;
5      stat(direntry->d_name, &data);          /* get stat data */
6      shmptr->tot_size += data.st_size;
7      shmptr->tot_files++;
8      if (S_ISREG(data.st_mode)) shmptr->tot_regular++;
9      if (S_ISFIFO(data.st_mode)) shmptr->tot_fifo++;
10     if (S_ISLNK(data.st_mode)) shmptr->tot_link++;
11     if (S_ISDIR(data.st_mode)) shmptr->tot_dir++;
12     if (S_ISBLK(data.st_mode)) shmptr->tot_block++;
13     if (S_ISCHR(data.st_mode)) shmptr->tot_char++;
14     if (S_ISSOCK(data.st_mode)) shmptr->tot_sock++;
15     return 0;
16 }
17 /* Signal Handler to manage termination */
18 void HandSIGTERM(int signo) {
19     MutexLock(mutex);
20     ShmRemove(key, shmptr);
21     MutexRemove(mutex);
22     exit(0);
23 }

```

Figura 11.26: Codice delle funzioni ausiliarie usate da `DirMonitor.c`.

Il codice di quest'ultima è riportato in fig. 11.26. Come si vede la funzione (2-16) è molto semplice e si limita a chiamare (5) la funzione `stat` sul file indicato da ciascuna voce, per ottenerne i dati, che poi utilizza per incrementare i vari contatori nella memoria condivisa, cui accede grazie alla variabile globale `shmptr`.

Dato che la funzione è chiamata da `dir_scan`, si è all'interno del ciclo principale del programma, con un mutex acquisito, perciò non è necessario effettuare nessun controllo e si può accedere direttamente alla memoria condivisa usando `shmptr` per riempire i campi della

struttura `DirProp`; così prima (6-7) si sommano le dimensioni dei file ed il loro numero, poi, utilizzando le macro di tab. 4.6, si contano (8-14) quanti ce ne sono per ciascun tipo.

In fig. 11.26 è riportato anche il codice (17-23) del gestore dei segnali di terminazione, usato per chiudere il programma. Esso, oltre a provocare l'uscita del programma, si incarica anche di cancellare tutti gli oggetti di intercomunicazione non più necessari. Per questo anzitutto (19) acquisisce il mutex con `MutexLock`, per evitare di operare mentre un client sta ancora leggendo i dati, dopo di che (20) distacca e rimuove il segmento di memoria condivisa usando `ShmRemove`. Infine (21) rimuove il mutex con `MutexRemove` ed esce (22).

```

1 int main(int argc, char *argv[])
2 {
3     key_t key;
4     ...
5     /* create needed IPC objects */
6     key = ftok("~/gapil/sources/DirMonitor.c", 1); /* define a key */
7     if (!(shmptr = ShmFind(key, 4096)) { /* get a shared memory segment */
8         perror("Cannot find shared memory");
9         exit(1);
10    }
11    if ((mutex = MutexFind(key)) == -1) { /* get the Mutex */
12        perror("Cannot find mutex");
13        exit(1);
14    }
15    /* main loop */
16    MutexLock(mutex); /* lock shared memory */
17    printf("Ci sono %d file dati\n", shmptr->tot_regular);
18    printf("Ci sono %d directory\n", shmptr->tot_dir);
19    printf("Ci sono %d link\n", shmptr->tot_link);
20    printf("Ci sono %d fifo\n", shmptr->tot_fifo);
21    printf("Ci sono %d socket\n", shmptr->tot_sock);
22    printf("Ci sono %d device a caratteri\n", shmptr->tot_char);
23    printf("Ci sono %d device a blocchi\n", shmptr->tot_block);
24    printf("Totale %d file, %d per %d byte\n",
25           shmptr->tot_files, shmptr->tot_size);
26    MutexUnlock(mutex); /* unlock shared memory */
27 }

```

Figura 11.27: Codice del programma client del monitor delle proprietà di una directory, `ReadMonitor.c`.

Il codice del client usato per leggere le informazioni mantenute nella memoria condivisa è riportato in fig. 11.27. Al solito si è omessa la sezione di gestione delle opzioni e la funzione che stampa a video le istruzioni; il codice completo è nei sorgenti allegati, nel file `ReadMonitor.c`.

Una volta conclusa la gestione delle opzioni a riga di comando il programma rigenera (7) con `ftok` la stessa chiave usata dal server per identificare il segmento di memoria condivisa ed il mutex, poi (8) richiede con `ShmFind` l'indirizzo della memoria condivisa agganciando al contempo il segmento al processo. Infine (17-20) con `MutexFind` si richiede l'identificatore del mutex. Completata l'inizializzazione ed ottenuti i riferimenti agli oggetti di intercomunicazione necessari viene eseguito il corpo principale del programma (21-33); si comincia (22) acquisendo il mutex con `MutexLock`; qui avviene il blocco del processo se la memoria condivisa non è disponibile. Poi (23-31) si stampano i vari valori mantenuti nella memoria condivisa attraverso l'uso di `shmptr`. Infine (41) con `MutexUnlock` si rilascia il mutex, prima di uscire.

Verifichiamo allora il funzionamento dei nostri programmi; al solito, usando le funzioni di libreria occorre definire opportunamente `LD_LIBRARY_PATH`; poi si potrà lanciare il server con:

```
[piccardi@gont sources]$ ./dirmonitor ./
```

ed avendo usato **daemon** il comando ritornerà immediatamente. Una volta che il server è in esecuzione, possiamo passare ad invocare il client per verificarne i risultati, in tal caso otterremo:

```
[piccardi@gont sources]$ ./readmon
Ci sono 68 file dati
Ci sono 3 directory
Ci sono 0 link
Ci sono 0 fifo
Ci sono 0 socket
Ci sono 0 device a caratteri
Ci sono 0 device a blocchi
Totale 71 file, per 489831 byte
```

ed un rapido calcolo (ad esempio con `ls -a | wc` per contare i file) ci permette di verificare che il totale dei file è giusto. Un controllo con **ipcs** ci permette inoltre di verificare la presenza di un segmento di memoria condivisa e di un semaforo:

```
[piccardi@gont sources]$ ipcs
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status
0xfffffffff 54067205  piccardi   666        4096       1

----- Semaphore Arrays -----
key          semid      owner      perms      nsems
0xfffffffff 229376    piccardi   666        1

----- Message Queues -----
key          msqid      owner      perms      used-bytes  messages
```

Se a questo punto aggiungiamo un file, ad esempio con **touch prova**, potremo verificare che, passati nel peggiore dei casi almeno 10 secondi (o l'eventuale altro intervallo impostato per la riletture dei dati) avremo:

```
[piccardi@gont sources]$ ./readmon
Ci sono 69 file dati
Ci sono 3 directory
Ci sono 0 link
Ci sono 0 fifo
Ci sono 0 socket
Ci sono 0 device a caratteri
Ci sono 0 device a blocchi
Totale 72 file, per 489887 byte
```

A questo punto possiamo far uscire il server inviandogli un segnale di **SIGTERM** con il comando **killall dirmonitor**, a questo punto ripetendo la lettura, otterremo un errore:

```
[piccardi@gont sources]$ ./readmon
Cannot find shared memory: No such file or directory
```

e inoltre potremo anche verificare che anche gli oggetti di intercomunicazione visti in precedenza sono stati regolarmente cancellati:

```
[piccardi@gont sources]$ ipcs
----- Shared Memory Segments -----
key          shmid      owner      perms      bytes      nattch     status

----- Semaphore Arrays -----
key          semid      owner      perms      nsems

----- Message Queues -----
key          msqid      owner      perms      used-bytes  messages
```

11.3 Tecniche alternative

Come abbiamo detto in sez. 11.2.1, e ripreso nella descrizione dei singoli oggetti che ne fan parte, il *SysV-IPC* presenta numerosi problemi; in [?] ¹⁸ Stevens ne effettua una accurata analisi (alcuni dei concetti sono già stati accennati in precedenza) ed elenca alcune possibili tecniche alternative, che vogliamo riprendere in questa sezione.

11.3.1 Alternative alle code di messaggi

Le code di messaggi sono probabilmente il meno usato degli oggetti del *SysV-IPC*; esse infatti nacquero principalmente come meccanismo di comunicazione bidirezionale quando ancora le *pipe* erano unidirezionali; con la disponibilità di *socketpair* (vedi sez. 11.1.5) o utilizzando una coppia di *pipe*, si può ottenere questo risultato senza incorrere nelle complicazioni introdotte dal *SysV-IPC*.

In realtà, grazie alla presenza del campo *mtype*, le code di messaggi hanno delle caratteristiche ulteriori, consentendo una classificazione dei messaggi ed un accesso non rigidamente sequenziale; due caratteristiche che sono impossibili da ottenere con le *pipe* e i socket di *socketpair*. A queste esigenze però si può comunque ovviare in maniera diversa con un uso combinato della memoria condivisa e dei meccanismi di sincronizzazione, per cui alla fine l'uso delle code di messaggi classiche è relativamente poco diffuso.

11.3.2 I file di lock

Come illustrato in sez. 11.2.5 i semafori del *SysV-IPC* presentano una interfaccia inutilmente complessa e con alcuni difetti strutturali, per questo quando si ha una semplice esigenza di sincronizzazione per la quale basterebbe un semaforo binario (quello che abbiamo definito come *mutex*), per indicare la disponibilità o meno di una risorsa, senza la necessità di un contatore come i semafori, si possono utilizzare metodi alternativi.

La prima possibilità, utilizzata fin dalle origini di Unix, è quella di usare dei *file di lock* (per i quali è stata anche riservata una opportuna directory, */var/lock*, nella standardizzazione del *Filesystem Hierarchy Standard*). Per questo si usa la caratteristica della funzione *open* (illustrata in sez. 5.1.2) che prevede ¹⁹ che essa ritorni un errore quando usata con i flag di *O_CREAT* e *O_EXCL*. In tal modo la creazione di un *file di lock* può essere eseguita atomicamente, il processo che crea il file con successo si può considerare come titolare del lock (e della risorsa ad esso associata) mentre il rilascio si può eseguire con una chiamata ad *unlink*.

Un esempio dell'uso di questa funzione è mostrato dalle funzioni *LockFile* ed *UnlockFile* riportate in fig. 11.28 (sono contenute in *LockFile.c*, un altro dei sorgenti allegati alla guida) che permettono rispettivamente di creare e rimuovere un *file di lock*. Come si può notare entrambe le funzioni sono elementari; la prima (4-10) si limita ad aprire il file di lock (9) nella modalità descritta, mentre la seconda (11-17) lo cancella con *unlink*.

Uno dei limiti di questa tecnica è che, come abbiamo già accennato in sez. 5.1.2, questo comportamento di *open* può non funzionare (la funzione viene eseguita, ma non è garantita l'atomicità dell'operazione) se il filesystem su cui si va ad operare è su NFS; in tal caso si può adottare una tecnica alternativa che prevede l'uso della *link* per creare come *file di lock* un hard link ad un file esistente; se il link esiste già e la funzione fallisce, significa che la risorsa è bloccata e potrà essere sbloccata solo con un *unlink*, altrimenti il link è creato ed il lock

¹⁸in particolare nel capitolo 14.

¹⁹questo è quanto dettato dallo standard POSIX.1, ciò non toglie che in alcune implementazioni questa tecnica possa non funzionare; in particolare per Linux, nel caso di NFS, si è comunque soggetti alla possibilità di una *race condition*.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <unistd.h>      /* Unix standard functions */
4 /*
5  * Function LockFile:
6  */
7 int LockFile(const char* path_name)
8 {
9     return open(path_name, O_EXCL|O_CREAT);
10 }
11 /*
12 * Function UnlockFile:
13 */
14 int UnlockFile(const char* path_name)
15 {
16     return unlink(path_name);
17 }
```

Figura 11.28: Il codice delle funzioni LockFile e UnlockFile che permettono di creare e rimuovere un *file di lock*.

acquisito; il controllo e l'eventuale acquisizione sono atomici; la soluzione funziona anche su NFS, ma ha un altro difetto è che è quello di poterla usare solo se si opera all'interno di uno stesso filesystem.

In generale comunque l'uso di un *file di lock* presenta parecchi problemi che non lo rendono una alternativa praticabile per la sincronizzazione: anzitutto in caso di terminazione imprevista del processo, si lascia allocata la risorsa (il *file di lock*) e questa deve essere sempre cancellata esplicitamente. Inoltre il controllo della disponibilità può essere eseguito solo con una tecnica di *polling*, ed è quindi molto inefficiente.

La tecnica dei file di lock ha comunque una sua utilità, e può essere usata con successo quando l'esigenza è solo quella di segnalare l'occupazione di una risorsa, senza necessità di attendere che questa si liberi; ad esempio la si usa spesso per evitare interferenze sull'uso delle porte seriali da parte di più programmi: qualora si trovi un file di lock il programma che cerca di accedere alla seriale si limita a segnalare che la risorsa non è disponibile.

11.3.3 La sincronizzazione con il *file locking*

Dato che i file di lock presentano gli inconvenienti illustrati in precedenza, la tecnica alternativa di sincronizzazione più comune è quella di fare ricorso al *file locking* (trattato in sez. 10.1) usando `fcntl` su un file creato per l'occasione per ottenere un write lock. In questo modo potremo usare il lock come un *mutex*: per bloccare la risorsa basterà acquisire il lock, per sbloccarla basterà rilasciare il lock. Una richiesta fatta con un write lock metterà automaticamente il processo in stato di attesa, senza necessità di ricorrere al *polling* per determinare la disponibilità della risorsa, e al rilascio della stessa da parte del processo che la occupava si otterrà il nuovo lock atomicamente.

Questo approccio presenta il notevole vantaggio che alla terminazione di un processo tutti i lock acquisiti vengono rilasciati automaticamente (alla chiusura dei relativi file) e non ci si deve preoccupare di niente; inoltre non consuma risorse permanentemente allocate nel sistema. Lo svantaggio è che, dovendo fare ricorso a delle operazioni sul filesystem, esso è in genere leggermente più lento.

```

1  /* Function CreateMutex: Create a mutex using file locking. */
2  int CreateMutex(const char *path_name)
3  {
4      return open(path_name, O_EXCL|O_CREAT);
5  }
6  /* Function UnlockMutex: unlock a file. */
7  int FindMutex(const char *path_name)
8  {
9      return open(path_name, O_RDWR);
10 }
11 /* Function LockMutex: lock mutex using file locking. */
12 int LockMutex(int fd)
13 {
14     struct flock lock;          /* file lock structure */
15     lock.l_type = F_WRLCK;      /* set type: read or write */
16     lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
17     lock.l_start = 0;           /* set the start of the locked region */
18     lock.l_len = 0;             /* set the length of the locked region */
19     /* do locking */
20     return fcntl(fd, F_SETLKW, &lock);
21 }
22 /* Function UnlockMutex: unlock a file. */
23 int UnlockMutex(int fd)
24 {
25     struct flock lock;          /* file lock structure */
26     lock.l_type = F_UNLCK;      /* set type: unlock */
27     lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
28     lock.l_start = 0;           /* set the start of the locked region */
29     lock.l_len = 0;             /* set the length of the locked region */
30     /* do locking */
31     return fcntl(fd, F_SETLK, &lock);
32 }
33 /* Function RemoveMutex: remove a mutex (unlinking the lock file). */
34 int RemoveMutex(const char *path_name)
35 {
36     return unlink(path_name);
37 }
38 /* Function ReadMutex: read a mutex status. */
39 int ReadMutex(int fd)
40 {
41     int res;
42     struct flock lock;          /* file lock structure */
43     lock.l_type = F_WRLCK;      /* set type: unlock */
44     lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
45     lock.l_start = 0;           /* set the start of the locked region */
46     lock.l_len = 0;             /* set the length of the locked region */
47     /* do locking */
48     if ( (res = fcntl(fd, F_GETLK, &lock)) ) {
49         return res;
50     }
51     return lock.l_type;
52 }

```

Figura 11.29: Il codice delle funzioni che permettono per la gestione dei *mutex* con il *file locking*.

Il codice delle varie funzioni usate per implementare un mutex utilizzando il *file locking* è riportato in fig. 11.29; si è mantenuta volutamente una struttura analoga alle precedenti funzioni che usano i semafori, anche se le due interfacce non possono essere completamente equivalenti, specie per quanto riguarda la rimozione del mutex.

La prima funzione (1-5) è `CreateMutex`, e serve a creare il mutex; la funzione è estremamente semplice, e si limita (4) a creare, con una opportuna chiamata ad `open`, il file che sarà usato per il successivo *file locking*, assicurandosi che non esista già (nel qual caso segnala un errore); poi restituisce il file descriptor che sarà usato dalle altre funzioni per acquisire e rilasciare il mutex.

La seconda funzione (6-10) è `FindMutex`, che, come la precedente, è stata definita per mantenere una analogia con la corrispondente funzione basata sui semafori. Anch'essa si limita (9) ad aprire il file da usare per il *file locking*, solo che in questo caso le opzioni di `open` sono tali che il file in questione deve esistere di già.

La terza funzione (11-22) è `LockMutex` e serve per acquisire il mutex. La funzione definisce (14) e inizializza (16-19) la struttura `lock` da usare per acquisire un write lock sul file, che poi (21) viene richiesto con `fcntl`, restituendo il valore di ritorno di quest'ultima. Se il file è libero il lock viene acquisito e la funzione ritorna immediatamente; altrimenti `fcntl` si bloccherà (si noti che la si è chiamata con `F_SETLKW`) fino al rilascio del lock.

La quarta funzione (24-34) è `UnlockMutex` e serve a rilasciare il mutex. La funzione è analoga alla precedente, solo che in questo caso si inizializza (28-31) la struttura `lock` per il rilascio del lock, che viene effettuato (33) con la opportuna chiamata a `fcntl`. Avendo usato il *file locking* in semantica POSIX (si riveda quanto detto sez. 10.1.3) solo il processo che ha precedentemente eseguito il lock può sbloccare il mutex.

La quinta funzione (36-39) è `RemoveMutex` e serve a cancellare il mutex. Anche questa funzione è stata definita per mantenere una analogia con le funzioni basate sui semafori, e si limita a cancellare (38) il file con una chiamata ad `unlink`. Si noti che in questo caso la funzione non ha effetto sui mutex già ottenuti con precedenti chiamate a `FindMutex` o `CreateMutex`, che continueranno ad essere disponibili fintanto che i relativi file descriptor restano aperti. Pertanto per rilasciare un mutex occorrerà prima chiamare `UnlockMutex` oppure chiudere il file usato per il lock.

La sesta funzione (41-55) è `ReadMutex` e serve a leggere lo stato del mutex. In questo caso si prepara (46-49) la solita struttura `lock` come l'acquisizione del lock, ma si effettua (51) la chiamata a `fcntl` usando il comando `F_GETLK` per ottenere lo stato del lock, e si restituisce (52) il valore di ritorno in caso di errore, ed il valore del campo `l_type` (che descrive lo stato del lock) altrimenti (54). Per questo motivo la funzione restituirà -1 in caso di errore e uno dei due valori `F_UNLCK` o `F_WRLCK`²⁰ in caso di successo, ad indicare che il mutex è, rispettivamente, libero o occupato.

Basandosi sulla semantica dei file lock POSIX valgono tutte le considerazioni relative al comportamento di questi ultimi fatte in sez. 10.1.3; questo significa ad esempio che, al contrario di quanto avveniva con l'interfaccia basata sui semafori, chiamate multiple a `UnlockMutex` o `LockMutex` non si cumulano e non danno perciò nessun inconveniente.

11.3.4 Il *memory mapping* anonimo

Abbiamo già visto che quando i processi sono *correlati*, se cioè hanno almeno un progenitore comune, l'uso delle *pipe* può costituire una valida alternativa alle code di messaggi; nella stessa

²⁰non si dovrebbe mai avere il terzo valore possibile, `F_RDLCK`, dato che la nostra interfaccia usa solo i write lock. Però è sempre possibile che siano richiesti altri lock sul file al di fuori dell'interfaccia, nel qual caso si potranno avere, ovviamente, interferenze indesiderate.

situazione si può evitare l'uso di una memoria condivisa facendo ricorso al cosiddetto *memory mapping* anonimo.

In sez. 10.4.1 abbiamo visto come sia possibile mappare il contenuto di un file nella memoria di un processo, e che, quando viene usato il flag `MAP_SHARED`, le modifiche effettuate al contenuto del file vengono viste da tutti i processi che lo hanno mappato. Utilizzare questa tecnica per creare una memoria condivisa fra processi diversi è estremamente inefficiente, in quanto occorre passare attraverso il disco.

Però abbiamo visto anche che se si esegue la mappatura con il flag `MAP_ANONYMOUS` la regione mappata non viene associata a nessun file, anche se quanto scritto rimane in memoria e può essere riletto; allora, dato che un processo figlio mantiene nel suo spazio degli indirizzi anche le regioni mappate, esso sarà anche in grado di accedere a quanto in esse è contenuto.

In questo modo diventa possibile creare una memoria condivisa fra processi diversi, purché questi abbiano almeno un progenitore comune che ha effettuato il *memory mapping* anonimo.²¹ Vedremo come utilizzare questa tecnica più avanti, quando realizzeremo una nuova versione del monitor visto in sez. 11.2.6 che possa restituire i risultati via rete.

11.4 L'intercomunicazione fra processi di POSIX

Per superare i numerosi problemi del *SysV-IPC*, evidenziati per i suoi aspetti generali in coda a sez. 11.2.1 e per i singoli oggetti nei paragrafi successivi, lo standard POSIX.1b ha introdotto dei nuovi meccanismi di comunicazione, che vanno sotto il nome di POSIX IPC, definendo una interfaccia completamente nuova, che tratteremo in questa sezione.

11.4.1 Considerazioni generali

Oggi Linux supporta tutti gli oggetti definiti nello standard POSIX per l'IPC, ma a lungo non è stato così; la memoria condivisa è presente a partire dal kernel 2.4.x, i semafori sono forniti dalla *glibc* nella sezione che implementa i *thread* POSIX di nuova generazione che richiedono il kernel 2.6, le code di messaggi sono supportate a partire dal kernel 2.6.6.

La caratteristica fondamentale dell'interfaccia POSIX è l'abbandono dell'uso degli identificatori e delle chiavi visti nel *SysV-IPC*, per passare ai *POSIX IPC names*, che sono sostanzialmente equivalenti ai nomi dei file. Tutte le funzioni che creano un oggetto di IPC POSIX prendono come primo argomento una stringa che indica uno di questi nomi; lo standard è molto generico riguardo l'implementazione, ed i nomi stessi possono avere o meno una corrispondenza sul filesystem; tutto quello che è richiesto è che:

- i nomi devono essere conformi alle regole che caratterizzano i *pathname*, in particolare non essere più lunghi di `PATH_MAX` byte e terminati da un carattere nullo.
- se il nome inizia per una `/` chiamate differenti allo stesso nome fanno riferimento allo stesso oggetto, altrimenti l'interpretazione del nome dipende dall'implementazione.
- l'interpretazione di ulteriori `/` presenti nel nome dipende dall'implementazione.

Data la assoluta genericità delle specifiche, il comportamento delle funzioni è subordinato in maniera quasi completa alla relativa implementazione, tanto che Stevens in [?] cita questo caso come un esempio della maniera standard usata dallo standard POSIX per consentire implementazioni non standardizzabili.

²¹nei sistemi derivati da SysV una funzionalità simile a questa viene implementata mappando il file speciale `/dev/zero`. In tal caso i valori scritti nella regione mappata non vengono ignorati (come accade qualora si scriva direttamente sul file), ma restano in memoria e possono essere rilette secondo le stesse modalità usate nel *memory mapping* anonimo.

Nel caso di Linux, sia per quanto riguarda la memoria condivisa ed i semafori, che per le code di messaggi, tutto viene creato usando come radici delle opportune directory (rispettivamente `/dev/shm` e `/dev/mqueue`, per i dettagli si faccia riferimento a sez. 11.4.3, sez. 11.4.4 e sez. 11.4.2). I nomi specificati nelle relative funzioni devono essere nella forma di un *pathname* assoluto (devono cioè iniziare con `/`) e corrisponderanno ad altrettanti file creati all'interno di queste directory; per questo motivo detti nomi non possono contenere altre `/` oltre quella iniziale.

Il vantaggio degli oggetti di IPC POSIX è comunque che essi vengono inseriti nell'albero dei file, e possono essere maneggiati con le usuali funzioni e comandi di accesso ai file, che funzionano come su dei file normali. Questo però è vero nel caso di Linux, che usa una implementazione che lo consente, non è detto che altrettanto valga per altri kernel. In particolare, come si può facilmente verificare con il comando `strace`, sia per la memoria condivisa che per le code di messaggi varie *system call* utilizzate da Linux corrispondono in realtà a quelle ordinarie dei file, essendo detti oggetti realizzati come tali usando degli specifici filesystem.

In particolare i permessi associati agli oggetti di IPC POSIX sono identici ai permessi dei file, ed il controllo di accesso segue esattamente la stessa semantica (quella illustrata in sez. 4.4), e non quella particolare (si ricordi quanto visto in sez. 11.2.2) che viene usata per gli oggetti del SysV-IPC. Per quanto riguarda l'attribuzione dell'utente e del gruppo proprietari dell'oggetto alla creazione di quest'ultimo essa viene effettuata secondo la semantica SysV: corrispondono cioè a *UID* e *GID* effettivi del processo che esegue la creazione.

11.4.2 Code di messaggi Posix

Le code di messaggi POSIX sono supportate da Linux a partire dalla versione 2.6.6 del kernel. In generale, come le corrispettive del *SysV-IPC*, le code di messaggi sono poco usate, dato che i socket, nei casi in cui sono sufficienti, sono più comodi, e che in casi più complessi la comunicazione può essere gestita direttamente con mutex (o semafori) e memoria condivisa con tutta la flessibilità che occorre.

Per poter utilizzare le code di messaggi, oltre ad utilizzare un kernel superiore al 2.6.6 occorre utilizzare la libreria `librt` che contiene le funzioni dell'interfaccia POSIX ed i programmi che usano le code di messaggi devono essere compilati aggiungendo l'opzione `-lrt` al comando `gcc`. In corrispondenza all'inclusione del supporto nel kernel ufficiale le funzioni di libreria sono state inserite nella *glibc*, e sono disponibili a partire dalla versione 2.3.4 delle medesime.

La libreria inoltre richiede la presenza dell'apposito filesystem di tipo `mqueue` montato sulla directory `/dev/mqueue`; questo può essere fatto aggiungendo ad `/etc/fstab` una riga come:

				<code>/etc/fstab</code>					
<code>mqueue</code>	<code>/dev/mqueue</code>	<code>mqueue</code>	<code>defaults</code>	<code>0</code>	<code>0</code>				

ed esso sarà utilizzato come radice sulla quale vengono risolti i nomi delle code di messaggi che iniziano con una `/`. Le opzioni di mount accettate sono `uid`, `gid` e `mode` che permettono rispettivamente di impostare l'utente, il gruppo ed i permessi associati al filesystem.

La funzione di sistema che permette di aprire (e crearla se non esiste ancora) una coda di messaggi POSIX è `mq_open`, ed il suo prototipo è:

```
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag)
mqd_t mq_open(const char *name, int oflag, unsigned long mode, struct mq_attr *attr)
    Apre una coda di messaggi POSIX impostandone le caratteristiche.
```

La funzione ritorna il descrittore associato alla coda in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES	il processo non ha i privilegi per accedere alla coda secondo quanto specificato da <code>oflag</code> oppure <code>name</code> contiene più di una <code>"/</code> .
EEXIST	si è specificato <code>O_CREAT</code> e <code>O_EXCL</code> ma la coda già esiste.
EINVAL	il file non supporta la funzione, o si è specificato <code>O_CREAT</code> con un valore non nullo di <code>attr</code> e valori non validi dei campi <code>mq_maxmsg</code> e <code>mq_msgsize</code> ; questi valori devono essere positivi ed inferiori ai limiti di sistema se il processo non ha privilegi amministrativi, inoltre <code>mq_maxmsg</code> non può comunque superare <code>HARD_MAX</code> .
ENOENT	non si è specificato <code>O_CREAT</code> ma la coda non esiste o si è usato il nome <code>"/</code> .
ENOSPC	lo spazio è insufficiente, probabilmente per aver superato il limite di <code>queues_max</code> .

ed inoltre `EMFILE`, `ENAMETOOLONG`, `ENFILE`, `ENOMEM` ed nel loro significato generico.

La funzione apre la coda di messaggi identificata dall'argomento `name` restituendo il descrittore ad essa associato, del tutto analogo ad un file descriptor, con l'unica differenza che lo standard prevede un apposito tipo `mqd_t`. Nel caso di Linux si tratta in effetti proprio di un normale file descriptor; pertanto, anche se questo comportamento non è portabile, lo si può tenere sotto osservazione con le funzioni dell'I/O multiplexing (vedi sez. 10.2) come possibile alternativa all'uso dell'interfaccia di notifica di `mq_notify` (che vedremo a breve).

Se il nome indicato fa riferimento ad una coda di messaggi già esistente, il descrittore ottenuto farà riferimento allo stesso oggetto, pertanto tutti i processi che hanno usato `mq_open` su quel nome otterranno un riferimento alla stessa coda. Diventa così immediato costruire un canale di comunicazione fra detti processi.

La funzione è del tutto analoga ad `open` ed analoghi sono i valori che possono essere specificati per `oflag`, che deve essere specificato come maschera binaria; i valori possibili per i vari bit sono quelli visti in sez. 5.1.2 (per questo occorre includere `fcntl.h`) dei quali però `mq_open` riconosce solo i seguenti:

<code>O_RDONLY</code>	Apre la coda solo per la ricezione di messaggi. Il processo potrà usare il descrittore con <code>mq_receive</code> ma non con <code>mq_send</code> .
<code>O_WRONLY</code>	Apre la coda solo per la trasmissione di messaggi. Il processo potrà usare il descrittore con <code>mq_send</code> ma non con <code>mq_receive</code> .
<code>O_RDWR</code>	Apre la coda solo sia per la trasmissione che per la ricezione.
<code>O_CREAT</code>	Necessario qualora si debba creare la coda; la presenza di questo bit richiede la presenza degli ulteriori argomenti <code>mode</code> e <code>attr</code> .
<code>O_EXCL</code>	Se usato insieme a <code>O_CREAT</code> fa fallire la chiamata se la coda esiste già, altrimenti esegue la creazione atomicamente.
<code>O_NONBLOCK</code>	Imposta la coda in modalità non bloccante, le funzioni di ricezione e trasmissione non si bloccano quando non ci sono le risorse richieste, ma ritornano immediatamente con un errore di <code>EAGAIN</code> .

I primi tre bit specificano la modalità di apertura della coda, e sono fra loro esclusivi. Ma qualunque sia la modalità in cui si è aperta una coda, questa potrà essere riaperta più volte in

una modalità diversa, e vi si potrà sempre accedere attraverso descrittori diversi, esattamente come si può fare per i file normali.

Se la coda non esiste e la si vuole creare si deve specificare `O_CREAT`, in tal caso occorre anche specificare i permessi di creazione con l'argomento `mode`;²² i valori di quest'ultimo sono identici a quelli usati per `open` (per questo occorre includere `sys/stat.h`), anche se per le code di messaggi han senso solo i permessi di lettura e scrittura.

Oltre ai permessi di creazione possono essere specificati anche gli attributi specifici della coda tramite l'argomento `attr`; quest'ultimo è un puntatore ad una apposita struttura `mq_attr`, la cui definizione è riportata in fig. 11.30.

```

struct mq_attr {
    long    mq_flags;        /* message queue flags          */
    long    mq_maxmsg;       /* maximum number of messages   */
    long    mq_msgsize;      /* maximum message size         */
    long    mq_curmsgs;      /* number of messages currently */
};

```

Figura 11.30: La struttura `mq_attr`, contenente gli attributi di una coda di messaggi POSIX.

Per la creazione della coda i campi della struttura che devono essere specificati sono `mq_maxmsg` e `mq_msgsize`, che indicano rispettivamente il numero massimo di messaggi che può contenere e la dimensione massima di un messaggio. Il valore dovrà essere positivo e minore dei rispettivi limiti di sistema altrimenti la funzione fallirà con un errore di `EINVAL`. Se `attr` è un puntatore nullo gli attributi della coda saranno impostati ai valori predefiniti.

I suddetti limiti di sistema sono impostati attraverso una serie di file presenti sotto `/proc/sys/fs/mqueue`, in particolare i file che controllano i valori dei limiti sono:

`/proc/sys/fs/mqueue/msg_max`

Indica il valore massimo del numero di messaggi in una coda e agisce come limite superiore per il valore di `attr->mq_maxmsg` in `mq_open`. Il suo valore di default è 10. Il valore massimo è `HARD_MAX` che vale `(131072/sizeof(void *))`, ed il valore minimo 1 (ma era 10 per i kernel precedenti il 2.6.28). Questo limite viene ignorato per i processi con privilegi amministrativi (più precisamente con la *capability* `CAP_SYS_RESOURCE`) ma `HARD_MAX` resta comunque non superabile.

`/proc/sys/fs/mqueue/msgsize_max`

Indica il valore massimo della dimensione in byte di un messaggio sulla coda ed agisce come limite superiore per il valore di `attr->mq_msgsize` in `mq_open`. Il suo valore di default è 8192. Il valore massimo è 1048576 ed il valore minimo 128 (ma per i kernel precedenti il 2.6.28 detti limiti erano rispettivamente `INT_MAX` e 8192). Questo limite viene ignorato dai processi con privilegi amministrativi (con la *capability* `CAP_SYS_RESOURCE`).

`/proc/sys/fs/mqueue/queues_max`

Indica il numero massimo di code di messaggi creabili in totale sul sistema, il valore di default è 256 ma si può usare un valore qualunque fra 0 e `INT_MAX`. Il limite non viene applicato ai processi con privilegi amministrativi (cioè con la *capability* `CAP_SYS_RESOURCE`).

²²fino al 2.6.14 per un bug i valori della *umask* del processo non venivano applicati a questi permessi.

Infine sulle code di messaggi si applica il limite imposto sulla risorsa `RLIMIT_MSGQUEUE` (vedi sez. 6.3.2) che indica lo spazio massimo (in byte) occupabile dall'insieme di tutte le code di messaggi appartenenti ai processi di uno stesso utente, che viene identificato in base al *real user ID* degli stessi.

Quando l'accesso alla coda non è più necessario si può chiudere il relativo descrittore con la funzione `mq_close`, il cui prototipo è:

```
#include <mqqueue.h>
int mq_close(mqd_t mqdes)
```

Chiude una coda di messaggi.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori `EBADF` o `EINTR` nel loro significato generico.

La funzione è analoga a `close`,²³ dopo la sua esecuzione il processo non sarà più in grado di usare il descrittore della coda, ma quest'ultima continuerà ad esistere nel sistema e potrà essere acceduta con un'altra chiamata a `mq_open`. All'uscita di un processo tutte le code aperte, così come i file, vengono chiuse automaticamente. Inoltre se il processo aveva agganciato una richiesta di notifica sul descrittore che viene chiuso, questa sarà rilasciata e potrà essere richiesta da qualche altro processo.

Quando si vuole effettivamente rimuovere una coda dal sistema occorre usare la funzione di sistema `mq_unlink`, il cui prototipo è:

```
#include <mqqueue.h>
int mq_unlink(const char *name)
```

Rimuove una coda di messaggi.

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà gli uno dei valori:

`EACCES` non si hanno i permessi per cancellare la coda.

`ENAMETOOLONG` il nome indicato è troppo lungo.

`ENOENT` non esiste una coda con il nome indicato.

Anche in questo caso il comportamento della funzione è analogo a quello di `unlink` per i file, la funzione rimuove la coda `name` (ed il relativo file sotto `/dev/mqueue`), così che una successiva chiamata a `mq_open` fallisce o crea una coda diversa.

Come per i file ogni coda di messaggi ha un contatore di riferimenti, per cui la coda non viene effettivamente rimossa dal sistema fin quando questo non si annulla. Pertanto anche dopo aver eseguito con successo `mq_unlink` la coda resterà accessibile a tutti i processi che hanno un descrittore aperto su di essa. Allo stesso modo una coda ed i suoi contenuti resteranno disponibili all'interno del sistema anche quando quest'ultima non è aperta da nessun processo (questa è una delle differenze più rilevanti nei confronti di *pipe* e *fifo*). La sola differenza fra code di messaggi POSIX e file normali è che, essendo il filesystem delle code di messaggi virtuale, e basato su oggetti interni al kernel, il suo contenuto viene perduto con il riavvio del sistema.

Come accennato ad ogni coda di messaggi è associata una struttura `mq_attr`, che può essere letta e modificata attraverso le due funzioni `mq_getattr` e `mq_setattr`, i cui prototipi sono:

²³su Linux, dove le code sono implementate come file su un filesystem dedicato, è esattamente la stessa funzione, per cui non esiste una *system call* autonoma e la funzione viene rimappata su `close` dalla *glibc*.

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat)
    Legge gli attributi di una coda di messaggi POSIX.
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat, struct mq_attr *omqstat)
    Modifica gli attributi di una coda di messaggi POSIX.
```

Entrambe le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori `EBADF` o `EINVAL` nel loro significato generico.

La funzione `mq_getattr` legge i valori correnti degli attributi della coda `mqdes` nella struttura `mq_attr` puntata da `mqstat`; di questi l'unico relativo allo stato corrente della coda è `mq_curmsgs` che indica il numero di messaggi da essa contenuti, gli altri indicano le caratteristiche generali della stessa impostate in fase di apertura.

La funzione `mq_setattr` permette di modificare gli attributi di una coda (indicata da `mqdes`) tramite i valori contenuti nella struttura `mq_attr` puntata da `mqstat`, ma può essere modificato solo il campo `mq_flags`, gli altri campi vengono comunque ignorati.

In particolare i valori di `mq_maxmsg` e `mq_msgsize` possono essere specificati solo in fase di creazione della coda. Inoltre i soli valori possibili per `mq_flags` sono 0 e `O_NONBLOCK`, per cui alla fine la funzione può essere utilizzata solo per abilitare o disabilitare la modalità non bloccante. L'argomento `omqstat` viene usato, quando diverso da `NULL`, per specificare l'indirizzo di una struttura su cui salvare i valori degli attributi precedenti alla chiamata della funzione.

Per inserire messaggi su di una coda sono previste due funzioni di sistema, `mq_send` e `mq_timedsend`. In realtà su Linux la *system call* è soltanto `mq_timedsend`, mentre `mq_send` viene implementata come funzione di libreria che si appoggia alla precedente. Inoltre `mq_timedsend` richiede che sia definita la macro `_XOPEN_SOURCE` ad un valore pari ad almeno 600 o la macro `_POSIX_C_SOURCE` ad un valore uguale o maggiore di 200112L. I rispettivi prototipi sono:

```
#include <mqueue.h>
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio)
    Esegue l'inserimento di un messaggio su una coda.

#include <mqueue.h>
#include <time.h>
int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
    unsigned int msg_prio, const struct timespec *abs_timeout)
    Esegue l'inserimento di un messaggio su una coda entro un tempo specificato
```

Entrambe le funzioni ritornano 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` si è aperta la coda con `O_NONBLOCK`, e la coda è piena.
`EBADF` si è specificato un file descriptor non valido.
`EINVAL` si è specificato un valore nullo per `msg_len`, o un valore di `msg_prio` fuori dai limiti, o un valore non valido per `abs_timeout`.
`EMSGSIZE` la lunghezza del messaggio `msg_len` eccede il limite impostato per la coda.
`ETIMEDOUT` l'inserimento del messaggio non è stato effettuato entro il tempo stabilito (solo `mq_timedsend`).
ed inoltre `EBADF` e `EINTR` nel loro significato generico.

Entrambe le funzioni richiedono un puntatore ad un buffer in memoria contenente il testo del messaggio da inserire nella coda `mqdes` nell'argomento `msg_ptr`, e la relativa lunghezza in `msg_len`. Se quest'ultima eccede la dimensione massima specificata da `mq_msgsize` le funzioni ritornano immediatamente con un errore di `EMSGSIZE`.

L'argomento `msg_prio` indica la priorità dell'argomento che essendo definito come `unsigned int` è sempre un intero positivo. I messaggi di priorità maggiore vengono inseriti davanti a quelli di priorità inferiore, e quindi saranno rilette per primi. A parità del valore della priorità il messaggio sarà inserito in coda a tutti quelli che hanno la stessa priorità che quindi saranno

letti con la politica di una *fifo*. Il valore della priorità non può eccedere il limite di sistema `MQ_PRIO_MAX`, che al momento è pari a 32768.

Qualora la coda sia piena, entrambe le funzioni si bloccano, a meno che non sia stata selezionata in fase di apertura della stessa la modalità non bloccante o non si sia impostato il flag `O_NONBLOCK` sul file descriptor della coda, nel qual caso entrambe ritornano con un codice di errore di `EAGAIN`.

La sola differenza fra le due funzioni è che `mq_timedsend`, passato il tempo massimo impostato con l'argomento `abs_timeout`, ritorna con un errore di `ETIMEDOUT`, se invece il tempo è già scaduto al momento della chiamata e la coda è piena la funzione ritorna immediatamente. Il valore di `abs_timeout` deve essere specificato come tempo assoluto tramite una struttura `timespec` (vedi fig. 4.16) indicato in numero di secondi e nanosecondi a partire dal 1 gennaio 1970.

Come per l'inserimento, anche per l'estrazione dei messaggi da una coda sono previste due funzioni di sistema, `mq_receive` e `mq_timedreceive`. Anche in questo caso su Linux soltanto `mq_timedreceive` è effettivamente, una *system call* e per usarla devono essere definite le opportune macro come per `mq_timedsend`. I rispettivi prototipi sono:

```
#include <queue.h>
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)
                                     Effettua la ricezione di un messaggio da una coda.

#include <queue.h>
#include <time.h>
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
                       unsigned int *msg_prio, const struct timespec *abs_timeout)
                                     Riceve un messaggio da una coda entro un limite di tempo.
```

Entrambe le funzioni ritornano il numero di byte del messaggio in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` si è aperta la coda con `O_NONBLOCK`, e la coda è vuota.

`EINVAL` si è specificato un valore nullo per `msg_ptr`, o un valore non valido per `abs_timeout`.

`EMSGSIZE` la lunghezza del messaggio sulla coda eccede il valore `msg_len` specificato per la ricezione.

`ETIMEDOUT` la ricezione del messaggio non è stata effettuata entro il tempo stabilito.

ed inoltre `EBADF` o `EINTR` nel loro significato generico.

La funzione estrae dalla coda `mqdes` il messaggio a priorità più alta, o il più vecchio fra quelli della stessa priorità. Una volta ricevuto il messaggio viene tolto dalla coda e la sua dimensione viene restituita come valore di ritorno; si tenga presente che 0 è una dimensione valida e che la condizione di errore è indicata soltanto da un valore di `-1`.²⁴

Se la dimensione specificata da `msg_len` non è sufficiente a contenere il messaggio, entrambe le funzioni, al contrario di quanto avveniva nelle code di messaggi di SysV, ritornano un errore di `EMSGSIZE` senza estrarre il messaggio. È pertanto opportuno eseguire sempre una chiamata a `mq_getattr` prima di eseguire una ricezione, in modo da ottenere la dimensione massima dei messaggi sulla coda, per poter essere in grado di allocare dei buffer sufficientemente ampi per la lettura.

Se si specifica un puntatore per l'argomento `msg_prio` il valore della priorità del messaggio viene memorizzato all'indirizzo da esso indicato. Qualora non interessi usare la priorità dei messaggi si può specificare `NULL`, ed usare un valore nullo della priorità nelle chiamate a `mq_send`.

²⁴Stevens in [?] fa notare che questo è uno dei casi in cui vale ciò che lo standard *non* dice, una dimensione nulla infatti, pur non essendo citata, non viene proibita.

Si noti che con le code di messaggi POSIX non si ha la possibilità di selezionare quale messaggio estrarre con delle condizioni sulla priorità, a differenza di quanto avveniva con le code di messaggi di SysV che permettono invece la selezione in base al valore del campo `mtype`.

Qualora la coda sia vuota entrambe le funzioni si bloccano, a meno che non si sia selezionata la modalità non bloccante; in tal caso entrambe ritornano immediatamente con l'errore `EAGAIN`. Anche in questo caso la sola differenza fra le due funzioni è che la seconda non attende indefinitamente e passato il tempo massimo `abs_timeout` ritorna comunque con un errore di `ETIMEDOUT`.

Uno dei problemi sottolineati da Stevens in [?], comuni ad entrambe le tipologie di code messaggi, è che non è possibile per chi riceve identificare chi è che ha inviato il messaggio, in particolare non è possibile sapere da quale utente esso provenga. Infatti, in mancanza di un meccanismo interno al kernel, anche se si possono inserire delle informazioni nel messaggio, queste non possono essere credute, essendo completamente dipendenti da chi lo invia. Vedremo però come, attraverso l'uso del meccanismo di notifica, sia possibile superare in parte questo problema.

Una caratteristica specifica delle code di messaggi POSIX è la possibilità di usufruire di un meccanismo di notifica asincrono; questo può essere attivato usando la funzione `mq_notify`, il cui prototipo è:

```
#include <mqqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *notification)
                                     Attiva il meccanismo di notifica per una coda.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` il descrittore non fa riferimento ad una coda di messaggi.

`EBUSY` c'è già un processo registrato per la notifica.

`EINVAL` si è richiesto un meccanismo di notifica invalido o specificato nella notifica con i segnali il valore di un segnale non esistente.

ed inoltre `ENOMEM` nel suo significato generico.

Il meccanismo di notifica permette di segnalare in maniera asincrona ad un processo la presenza di dati sulla coda indicata da `mqdes`, in modo da evitare la necessità di bloccarsi nell'attesa. Per far questo un processo deve registrarsi con la funzione `mq_notify`, ed il meccanismo è disponibile per un solo processo alla volta per ciascuna coda.

Il comportamento di `mq_notify` dipende dai valori passati con l'argomento `notification`, che è un puntatore ad una apposita struttura `sigevent`, (definita in fig. 7.15) introdotta dallo standard POSIX.1b per gestire la notifica di eventi; per altri dettagli su di essa si può rivedere quanto detto in sez. 7.5.2 a proposito dell'uso della stessa struttura per la notifica delle scadenze dei *timer*.

Attraverso questa struttura si possono impostare le modalità con cui viene effettuata la notifica nel campo `sigev_notify`, che può assumere i valori di tab. 7.11; fra questi la pagina di manuale riporta soltanto i primi tre, ed inizialmente era possibile solo `SIGEV_SIGNAL`. Il metodo consigliato è quello di usare `SIGEV_SIGNAL` usando il campo `sigev_signo` per indicare il quale segnale deve essere inviato al processo. Inoltre il campo `sigev_value` è un puntatore ad una struttura `sigval` (definita in fig. 7.14) che permette di restituire al gestore del segnale un valore numerico o un indirizzo,²⁵ posto che questo sia installato nella forma estesa vista in sez. 7.4.3.

La funzione registra il processo chiamante per la notifica se `notification` punta ad una struttura `sigevent` opportunamente inizializzata, o cancella una precedente registrazione se è

²⁵per il suo uso si riveda la trattazione fatta in sez. 7.5.1 a proposito dei segnali *real-time*.

NULL. Dato che un solo processo alla volta può essere registrato, la funzione fallisce con EBUSY se c'è un altro processo già registrato. Questo significa anche che se si registra una notifica con SIGEV_NONE il processo non la riceverà, ma impedirà anche che altri possano registrarsi per poterlo fare. Si tenga presente inoltre che alla chiusura del descrittore associato alla coda (e quindi anche all'uscita del processo) ogni eventuale registrazione di notifica presente viene cancellata.

La notifica del segnale avviene all'arrivo di un messaggio in una coda vuota (cioè solo se sulla coda non ci sono messaggi) e se non c'è nessun processo bloccato in una chiamata a `mq_receive`, in questo caso infatti il processo bloccato ha la precedenza ed il messaggio gli viene immediatamente inviato, mentre per il meccanismo di notifica tutto funziona come se la coda fosse rimasta vuota.

Quando un messaggio arriva su una coda vuota al processo che si era registrato viene inviato il segnale specificato da `notification->sigev_signo`, e la coda diventa disponibile per una ulteriore registrazione. Questo comporta che se si vuole mantenere il meccanismo di notifica occorre ripetere la registrazione chiamando nuovamente `mq_notify` all'interno del gestore del segnale di notifica. A differenza della situazione simile che si aveva con i segnali non affidabili (l'argomento è stato affrontato in 7.1.2) questa caratteristica non configura una *race condition* perché l'invio di un segnale avviene solo se la coda è vuota; pertanto se si vuole evitare di correre il rischio di perdere eventuali ulteriori segnali inviati nel lasso di tempo che occorre per ripetere la richiesta di notifica basta avere cura di eseguire questa operazione prima di estrarre i messaggi presenti dalla coda.

L'invio del segnale di notifica avvalora alcuni campi di informazione restituiti al gestore attraverso la struttura `siginfo_t` (definita in fig. 7.9). In particolare `si_pid` viene impostato al valore del *PID* del processo che ha emesso il segnale, `si_uid` all'*user-ID* effettivo, `si_code` a `SI_MESGQ`, e `si_errno` a 0. Questo ci dice che, se si effettua la ricezione dei messaggi usando esclusivamente il meccanismo di notifica, è possibile ottenere le informazioni sul processo che ha inserito un messaggio usando un gestore per il segnale in forma estesa, di nuovo si faccia riferimento a quanto detto al proposito in sez. 7.4.3 e sez. 7.5.1.

11.4.3 Memoria condivisa

La memoria condivisa è stato il primo degli oggetti di IPC POSIX inserito nel kernel ufficiale; il supporto a questo tipo di oggetti è realizzato attraverso il filesystem `tmpfs`, uno speciale filesystem che mantiene tutti i suoi contenuti in memoria, che viene attivato abilitando l'opzione `CONFIG_TMPFS` in fase di compilazione del kernel.

Per potere utilizzare l'interfaccia POSIX per la memoria condivisa la *glibc* (le funzioni sono state introdotte con la versione 2.2) richiede di compilare i programmi con l'opzione `-lrt`; inoltre è necessario che in `/dev/shm` sia montato un filesystem `tmpfs`; questo di norma viene fatto aggiungendo una riga del tipo di:

		/etc/fstab			
tmpfs	/dev/shm	tmpfs	defaults	0	0

ad `/etc/fstab`. In realtà si può montare un filesystem `tmpfs` dove si vuole, per usarlo come RAM disk, con un comando del tipo:

```
mount -t tmpfs -o size=128M,nr_inodes=10k,mode=700 tmpfs /mytmpfs
```

Il filesystem riconosce, oltre quelle mostrate, le opzioni `uid` e `gid` che identificano rispettivamente utente e gruppo cui assegnarne la titolarità, e `nr_blocks` che permette di specificarne

la dimensione in blocchi, cioè in multipli di `PAGECACHE_SIZE` che in questo caso è l'unità di allocazione elementare.

La funzione che permette di aprire un segmento di memoria condivisa POSIX, ed eventualmente di crearlo se non esiste ancora, è `shm_open`; il suo prototipo è:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode)
```

Apre un segmento di memoria condivisa.

La funzione ritorna un file descriptor in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES non si hanno i permessi di aprire il segmento nella modalità scelta o si richiesto `O_TRUNC` per un segmento su cui non si ha il permesso di scrittura.

EINVAL si è utilizzato un nome non valido.

ed inoltre `EEXIST`, `EMFILE`, `ENAMETOOLONG`, `ENFILE` e `ENOENT` nello stesso significato che hanno per `open`.

La funzione apre un segmento di memoria condivisa identificato dal nome `name`. Come già spiegato in sez. 11.4.1 questo nome può essere specificato in forma standard solo facendolo iniziare per `"/` e senza ulteriori `"/`. Linux supporta comunque nomi generici, che verranno interpretati prendendo come radice `/dev/shm`.²⁶

La funzione è del tutto analoga ad `open` ed analoghi sono i valori che possono essere specificati per `oflag`, che deve essere specificato come maschera binaria comprendente almeno uno dei due valori `O_RDONLY` e `O_RDWR`; i valori possibili per i vari bit sono quelli visti in sez. 5.1.2 dei quali però `shm_open` riconosce solo i seguenti:

- O_RDONLY** Apre il file descriptor associato al segmento di memoria condivisa per l'accesso in sola lettura.
- O_RDWR** Apre il file descriptor associato al segmento di memoria condivisa per l'accesso in lettura e scrittura.
- O_CREAT** Necessario qualora si debba creare il segmento di memoria condivisa se esso non esiste; in questo caso viene usato il valore di `mode` per impostare i permessi, che devono essere compatibili con le modalità con cui si è aperto il file.
- O_EXCL** Se usato insieme a `O_CREAT` fa fallire la chiamata a `shm_open` se il segmento esiste già, altrimenti esegue la creazione atomicamente.
- O_TRUNC** Se il segmento di memoria condivisa esiste già, ne tronca le dimensioni a 0 byte.

In caso di successo la funzione restituisce un file descriptor associato al segmento di memoria condiviso con le stesse modalità di `open` viste in sez. 5.1.2. Inoltre sul file descriptor viene sempre impostato il flag `FD_CLOEXEC`. Chiamate effettuate da diversi processi usando lo stesso nome restituiranno file descriptor associati allo stesso segmento, così come, nel caso di file ordinari, essi sono associati allo stesso inode. In questo modo è possibile effettuare una chiamata ad `mmap` sul file descriptor restituito da `shm_open` ed i processi vedranno lo stesso segmento di memoria condivisa.

Quando il nome non esiste si può creare un nuovo segmento specificando `O_CREAT`; in tal caso il segmento avrà (così come i nuovi file) lunghezza nulla. Il nuovo segmento verrà creato

²⁶occorre pertanto evitare di specificare qualcosa del tipo `/dev/shm/nome` all'interno di `name`, perché questo comporta, da parte delle funzioni di libreria, il tentativo di accedere a `/dev/shm/dev/shm/nome`.

con i permessi indicati da `mode` (di cui vengono usati solo i 9 bit meno significativi, non si applicano pertanto i permessi speciali di sez. 4.4.2) filtrati dal valore dell'*umask* del processo. Come gruppo ed utente proprietario del segmento saranno presi quelli facenti parte del gruppo *effective* del processo chiamante.

Dato che un segmento di lunghezza nulla è di scarsa utilità, una volta che lo si è creato per impostarne la dimensione si dovrà poi usare `ftruncate` (vedi sez. 4.3.3) prima di mapparlo in memoria con `mmap`. Si tenga presente che una volta chiamata `mmap` si può chiudere il file descriptor ad esso associato (semplicemente con `close`), senza che la mappatura ne risenta, e che questa può essere rimossa usando `munmap`.

Come per i file, quando si vuole rimuovere completamente un segmento di memoria condivisa occorre usare la funzione `shm_unlink`, il cui prototipo è:

<pre>#include <sys/mman.h> int shm_unlink(const char *name)</pre>	Rimuove un segmento di memoria condivisa.
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EACCES non si è proprietari del segmento.</p> <p>ed inoltre ENAMETOOLONG e ENOENT, nel loro significato generico.</p>	

La funzione è del tutto analoga ad `unlink`, e si limita a cancellare il nome del segmento da `/dev/shm`, senza nessun effetto né sui file descriptor precedentemente aperti con `shm_open`, né sui segmenti già mappati in memoria; questi verranno cancellati automaticamente dal sistema solo con le rispettive chiamate a `close` e `munmap`. Una volta eseguita questa funzione però, qualora si richieda l'apertura di un segmento con lo stesso nome, la chiamata a `shm_open` fallirà, a meno di non aver usato `O_CREAT`, in quest'ultimo caso comunque si otterrà un file descriptor che fa riferimento ad un segmento distinto da eventuali precedenti.

Dato che i segmenti di memoria condivisa sono trattati come file del filesystem `tmpfs`, si possono usare su di essi, con lo stesso significato che assumono sui file ordinari, anche funzioni come quelle delle famiglie `fstat`, `fchown` e `fchmod`. Inoltre a partire dal kernel 2.6.19 per i permessi sono supportate anche le ACL illustrate in sez. 4.5.2.

Come esempio dell'uso delle funzioni attinenti ai segmenti di memoria condivisa POSIX, vediamo come è possibile riscrivere una interfaccia semplificata analoga a quella vista in fig. 11.24 per la memoria condivisa in stile SysV. Il codice completo, di cui si sono riportate le parti essenziali in fig. 11.31, è contenuto nel file `SharedMem.c` dei sorgenti allegati.

La prima funzione (1-24) è `CreateShm` che, dato un nome nell'argomento `name` crea un nuovo segmento di memoria condivisa, accessibile in lettura e scrittura, e ne restituisce l'indirizzo. Anzitutto si definiscono (8) i flag per la successiva (9) chiamata a `shm_open`, che apre il segmento in lettura e scrittura (creandolo se non esiste, ed uscendo in caso contrario) assegnandogli sul filesystem i permessi specificati dall'argomento `perm`.

In caso di errore (10-12) si restituisce un puntatore nullo, altrimenti si prosegue impostando (14) la dimensione del segmento con `ftruncate`. Di nuovo (15-16) si esce immediatamente restituendo un puntatore nullo in caso di errore. Poi si passa (18) a mappare in memoria il segmento con `mmap` specificando dei diritti di accesso corrispondenti alla modalità di apertura. Di nuovo si restituisce (19-21) un puntatore nullo in caso di errore, altrimenti si inizializza (22) il contenuto del segmento al valore specificato dall'argomento `fill` con `memset`, e se ne restituisce (23) l'indirizzo.

La seconda funzione (25-40) è `FindShm` che trova un segmento di memoria condiviso esistente, restituendone l'indirizzo. In questo caso si apre (31) il segmento con `shm_open` richiedendo che il segmento sia già esistente, in caso di errore (31-33) si ritorna immediatamente un pun-

```

1 /* Function CreateShm: Create a shared memory segment mapping it */
2 void * CreateShm(char * shm_name, off_t shm_size, mode_t perm, int fill)
3 {
4     void * shm_ptr;
5     int fd;
6     int flag;
7     /* first open the object, creating it if not existent */
8     flag = O_CREAT|O_EXCL|O_RDWR;
9     fd = shm_open(shm_name, flag, perm); /* get object file descriptor */
10    if (fd < 0) {
11        return NULL;
12    }
13    /* set the object size */
14    if (ftruncate(fd, shm_size)) {
15        return NULL;
16    }
17    /* map it in the process address space */
18    shm_ptr = mmap(NULL, shm_size, PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
19    if (shm_ptr == MAP_FAILED) {
20        return NULL;
21    }
22    memset((void *) shm_ptr, fill, shm_size); /* fill segment */
23    return shm_ptr;
24 }
25 /* Function FindShm: Find a POSIX shared memory segment */
26 void * FindShm(char * shm_name, off_t shm_size)
27 {
28     void * shm_ptr;
29     int fd; /* ID of the IPC shared memory segment */
30     /* find shared memory ID */
31     if ((fd = shm_open(shm_name, O_RDWR|O_EXCL, 0)) < 0) {
32         return NULL;
33     }
34     /* take the pointer to it */
35     shm_ptr = mmap(NULL, shm_size, PROT_WRITE|PROT_READ, MAP_SHARED, fd, 0);
36     if (shm_ptr == MAP_FAILED) {
37         return NULL;
38     }
39     return shm_ptr;
40 }
41 /* Function RemoveShm: Remove a POSIX shared memory segment */
42 int RemoveShm(char * shm_name)
43 {
44     return shm_unlink(shm_name);
45 }

```

Figura 11.31: Il codice delle funzioni di gestione dei segmenti di memoria condivisa POSIX.

tatore nullo. Ottenuto il file descriptor del segmento lo si mappa (35) in memoria con `mmap`, restituendo (36-38) un puntatore nullo in caso di errore, o l'indirizzo (39) dello stesso in caso di successo.

La terza funzione (40-45) è `RemoveShm`, e serve a cancellare un segmento di memoria condivisa. Dato che al contrario di quanto avveniva con i segmenti del *SysV-IPC* gli oggetti allocati nel kernel vengono rilasciati automaticamente quando nessuna li usa più, tutto quello che c'è da fare (44) in questo caso è chiamare `shm_unlink`, restituendo al chiamante il valore di ritorno.

11.4.4 Semafori

Fino alla serie 2.4.x del kernel esisteva solo una implementazione parziale dei semafori POSIX che li realizzava solo a livello di *thread* e non di processi,²⁷ fornita attraverso la sezione delle estensioni *real-time* della *glibc* (quelle che si accedono collegandosi alla libreria *librt*). Esisteva inoltre una libreria che realizzava (parzialmente) l'interfaccia POSIX usando le funzioni dei semafori di *SysV-IPC* (mantenendo così tutti i problemi sottolineati in sez. 11.2.5).

A partire dal kernel 2.5.7 è stato introdotto un meccanismo di sincronizzazione completamente nuovo, basato sui cosiddetti *futex* (la sigla sta per *fast user mode mutex*) con il quale è stato possibile implementare una versione nativa dei semafori POSIX. Grazie a questo con i kernel della serie 2.6 e le nuove versioni della *glibc* che usano questa nuova infrastruttura per quella che viene chiamata *New Posix Thread Library*, sono state implementate anche tutte le funzioni dell'interfaccia dei semafori POSIX.

Anche in questo caso è necessario appoggiarsi alla libreria per le estensioni *real-time* *librt*, questo significa che se si vuole utilizzare questa interfaccia, oltre ad utilizzare gli opportuni file di definizione, occorrerà compilare i programmi con l'opzione `-lrt` o con `-lpthread` se si usano questi ultimi.

La funzione che permette di creare un nuovo semaforo POSIX, creando il relativo file, o di accedere ad uno esistente, è `sem_open`, questa prevede due forme diverse a seconda che sia utilizzata per aprire un semaforo esistente o per crearne uno nuovo, i relativi prototipi sono:

```
#include <semaphore.h>
#include <sys/stat.h>
#include <fcntl.h>
sem_t *sem_open(const char *name, int oflag)
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)
```

Crea un semaforo o ne apre uno esistente.

La funzione ritorna l'indirizzo del semaforo in caso di successo e `SEM_FAILED` per un errore, nel qual caso `errno` assumerà uno dei valori:

`EACCES` il semaforo esiste ma non si hanno permessi sufficienti per accedervi.
`EEXIST` si sono specificati `O_CREAT` e `O_EXCL` ma il semaforo esiste.
`EINVAL` il valore di `value` eccede `SEM_VALUE_MAX` o il nome è solo `"/`.
`ENAMETOOLONG` si è utilizzato un nome troppo lungo.
`ENOENT` non si è usato `O_CREAT` ed il nome specificato non esiste.
ed inoltre `EMFILE`, `ENFILE` ed `ENOMEM` nel loro significato generico.

L'argomento `name` definisce il nome del semaforo che si vuole utilizzare, ed è quello che permette a processi diversi di accedere allo stesso semaforo. Questo deve essere specificato nella stessa forma utilizzata per i segmenti di memoria condivisa, con un nome che inizia con `"/` e senza ulteriori `"/`, vale a dire nella forma `/nome-semaforo`.

Con Linux i file associati ai semafori sono mantenuti nel filesystem virtuale `/dev/shm`, e gli viene assegnato automaticamente un nome nella forma `sem.nome-semaforo`, si ha cioè una corrispondenza per cui `/nome-semaforo` viene rimappato, nella creazione tramite `sem_open`, su `/dev/shm/sem.nome-semaforo`. Per questo motivo la dimensione massima per il nome di un semaforo, a differenza di quanto avviene per i segmenti di memoria condivisa, è pari a `NAME_MAX-4`.

L'argomento `oflag` è quello che controlla le modalità con cui opera la funzione, ed è passato come maschera binaria; i bit corrispondono a quelli utilizzati per l'analogo argomento di `open`, anche se dei possibili valori visti in sez. 5.1.2 sono utilizzati soltanto `O_CREAT` e `O_EXCL`.

²⁷questo significava che i semafori erano visibili solo all'interno dei *thread* creati da un singolo processo, e non potevano essere usati come meccanismo di sincronizzazione fra processi diversi.

Se si usa `O_CREAT` si richiede la creazione del semaforo qualora questo non esista, ed in tal caso occorre utilizzare la seconda forma della funzione, in cui si devono specificare sia un valore iniziale con l'argomento `value`,²⁸ che una maschera dei permessi con l'argomento `mode`; se il semaforo esiste già questi saranno semplicemente ignorati. Usando il flag `O_EXCL` si richiede invece la verifica che il semaforo non esista, ed usandolo insieme ad `O_CREAT` la funzione fallisce qualora un semaforo con lo stesso nome sia già presente.

Si tenga presente che, come accennato in sez. 11.4.1, i semafori usano la semantica standard dei file per quanto riguarda i controlli di accesso, questo significa che un nuovo semaforo viene sempre creato con l'`UID` ed il `GID` effettivo del processo chiamante, e che i permessi indicati con `mode` vengono filtrati dal valore della `umask` del processo. Inoltre per poter aprire un semaforo è necessario avere su di esso sia il permesso di lettura che quello di scrittura.

La funzione restituisce in caso di successo un puntatore all'indirizzo del semaforo con un valore di tipo `sem_t *`, è questo valore che dovrà essere passato alle altre funzioni per operare sul semaforo stesso, e non sarà più necessario fare riferimento al nome, che potrebbe anche essere rimosso con `sem_unlink`.

Una volta che si sia ottenuto l'indirizzo di un semaforo, sarà possibile utilizzarlo; se si ricorda quanto detto all'inizio di sez. 11.2.5, dove si sono introdotti i concetti generali relativi ai semafori, le operazioni principali sono due, quella che richiede l'uso di una risorsa bloccando il semaforo e quella che rilascia la risorsa liberando il semaforo. La prima operazione è effettuata dalla funzione `sem_wait`, il cui prototipo è:

```
#include <semaphore.h>
int sem_wait(sem_t *sem)
```

Blocca un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINTR` la funzione è stata interrotta da un segnale.

`EINVAL` il semaforo `sem` non esiste.

La funzione cerca di decrementare il valore del semaforo indicato dal puntatore `sem`, se questo ha un valore positivo, cosa che significa che la risorsa è disponibile, la funzione ha successo, il valore del semaforo viene diminuito di 1 ed essa ritorna immediatamente consentendo la prosecuzione del processo.

Se invece il valore è nullo la funzione si blocca (fermando l'esecuzione del processo) fintanto che il valore del semaforo non ritorna positivo (cosa che a questo punto può avvenire solo per opera di altro processo che rilascia il semaforo con una chiamata a `sem_post`) così che poi essa possa decrementarlo con successo e proseguire.

Si tenga presente che la funzione può sempre essere interrotta da un segnale, nel qual caso si avrà un errore di `EINTR`; inoltre questo avverrà comunque, anche qualora si fosse richiesta la gestione con la semantica BSD, installando il gestore del suddetto segnale con l'opzione `SA_RESTART` (vedi sez. 7.4.3) per riavviare le *system call* interrotte.

Della funzione `sem_wait` esistono due varianti che consentono di gestire diversamente le modalità di attesa in caso di risorsa occupata, la prima di queste è `sem_trywait`, che serve ad effettuare un tentativo di acquisizione senza bloccarsi; il suo prototipo è:

```
#include <semaphore.h>
int sem_trywait(sem_t *sem)
```

Tenta di bloccare un semaforo.

²⁸e si noti come così diventa possibile, differenza di quanto avviene per i semafori del *SysV-IPC*, effettuare in maniera atomica creazione ed inizializzazione di un semaforo usando una unica funzione.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN il semaforo non può essere acquisito senza bloccarsi.

EINVAL l'argomento `sem` non indica un semaforo valido.

La funzione è identica a `sem_wait` ed se la risorsa è libera ha lo stesso effetto, vale a dire che in caso di semaforo diverso da zero la funzione lo decrementa e ritorna immediatamente; la differenza è che nel caso in cui il semaforo è occupato essa non si blocca e di nuovo ritorna immediatamente, restituendo però un errore di EAGAIN, così che il programma possa proseguire.

La seconda variante di `sem_wait` è una estensione specifica che può essere utilizzata soltanto se viene definita la macro `_XOPEN_SOURCE` ad un valore di almeno 600 o la macro `_POSIX_C_SOURCE` ad un valore uguale o maggiore di 200112L prima di includere `semaphore.h`, la funzione è `sem_timedwait`, ed il suo prototipo è:

```
#include <semaphore.h>
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout)
```

Blocca un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINTR la funzione è stata interrotta da un segnale.

EINVAL l'argomento `sem` non indica un semaforo valido.

ETIMEDOUT è scaduto il tempo massimo di attesa.

Anche in questo caso il comportamento della funzione è identico a quello di `sem_wait`, ma è possibile impostare un tempo limite per l'attesa tramite la struttura `timespec` (vedi fig. 4.16) puntata dall'argomento `abs_timeout`, indicato in secondi e nanosecondi a partire dalla cosiddetta *Epoch* (00:00:00, 1 January 1970 UTC). Scaduto il limite la funzione ritorna anche se non è possibile acquisire il semaforo fallendo con un errore di ETIMEDOUT.

La seconda funzione principale utilizzata per l'uso dei semafori è quella che viene usata per rilasciare un semaforo occupato o, in generale, per aumentare di una unità il valore dello stesso anche qualora non fosse occupato (si ricordi che in generale un semaforo viene usato come indicatore di un numero di risorse disponibili). Detta funzione è `sem_post` ed il suo prototipo è:

```
#include <semaphore.h>
int sem_post(sem_t *sem)
```

Rilascia un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EINVAL l'argomento `sem` non indica un semaforo valido.

OVERFLOW si superato il massimo valore di un semaforo.

La funzione incrementa di uno il valore corrente del semaforo indicato dall'argomento `sem`, se questo era nullo la relativa risorsa risulterà sbloccata, cosicché un altro processo (o *thread*) eventualmente bloccato in una `sem_wait` sul semaforo possa essere svegliato e rimesso in esecuzione. Si tenga presente che la funzione è sicura per l'uso all'interno di un gestore di segnali (si ricordi quanto detto in sez. 7.4.5).

Se invece di operare su un semaforo se ne volesse semplicemente leggere il valore, si potrà usare la funzione `sem_getvalue`, il cui prototipo è:

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval)
```

Richiede il valore di un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` l'argomento `sem` non indica un semaforo valido.

La funzione legge il valore del semaforo indicato dall'argomento `sem` e lo restituisce nella variabile intera puntata dall'argomento `sval`. Qualora ci siano uno o più processi bloccati in attesa sul semaforo lo standard prevede che la funzione possa restituire un valore nullo oppure il numero di processi bloccati in una `sem_wait` sul suddetto semaforo; nel caso di Linux vale la prima opzione.

Questa funzione può essere utilizzata per avere un suggerimento sullo stato di un semaforo, ovviamente non si può prendere il risultato riportato in `sval` che come indicazione, il valore del semaforo infatti potrebbe essere già stato modificato al ritorno della funzione.

Una volta che non ci sia più la necessità di operare su un semaforo se ne può terminare l'uso con la funzione `sem_close`, il cui prototipo è:

```
#include <semaphore.h>
int sem_close(sem_t *sem)
```

Chiude un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` l'argomento `sem` non indica un semaforo valido.

La funzione chiude il semaforo indicato dall'argomento `sem`, che non potrà più essere utilizzato nelle altre funzioni. La chiusura comporta anche che tutte le risorse che il sistema poteva avere assegnato al processo nell'uso del semaforo vengono immediatamente rilasciate. Questo significa che un eventuale altro processo bloccato sul semaforo a causa della acquisizione dello stesso da parte del processo che chiama `sem_close` potrà essere immediatamente riavviato.

Si tenga presente poi che come avviene per i file, all'uscita di un processo anche tutti i semafori che questo aveva aperto vengono automaticamente chiusi. Questo comportamento risolve il problema che si aveva con i semafori del *SysV IPC* (di cui si è parlato in sez. 11.2.5) per i quali le risorse possono restare bloccate. Si tenga infine presente che, a differenza di quanto avviene per i file, in caso di una chiamata ad `execve` tutti i semafori vengono chiusi automaticamente.

Come per i semafori del *SysV-IPC* anche quelli POSIX hanno una persistenza di sistema; questo significa che una volta che si è creato un semaforo con `sem_open` questo continuerà ad esistere fintanto che il kernel resta attivo (vale a dire fino ad un successivo riavvio) a meno che non lo si cancelli esplicitamente. Per far questo si può utilizzare la funzione `sem_unlink`, il cui prototipo è:

```
#include <semaphore.h>
int sem_unlink(const char *name)
```

Rimuove un semaforo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EACCES` non si hanno i permessi necessari a cancellare il semaforo.

`ENAMETOOLONG` il nome indicato è troppo lungo.

`ENOENT` il semaforo `name` non esiste.

La funzione rimuove il semaforo indicato dall'argomento `name`, che prende un valore identico a quello usato per creare il semaforo stesso con `sem_open`. Il semaforo viene rimosso dal

filesystem immediatamente; ma il semaforo viene effettivamente cancellato dal sistema soltanto quando tutti i processi che lo avevano aperto lo chiudono. Si segue cioè la stessa semantica usata con `unlink` per i file, trattata in dettaglio in sez. 4.2.1.

Una delle caratteristiche peculiari dei semafori POSIX è che questi possono anche essere utilizzati anche in forma anonima, senza necessità di fare ricorso ad un nome sul filesystem o ad altri indicativi. In questo caso si dovrà porre la variabile che contiene l'indirizzo del semaforo in un tratto di memoria che sia accessibile a tutti i processi in gioco. La funzione che consente di inizializzare un semaforo anonimo è `sem_init`, il cui prototipo è:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

Inizializza un semaforo anonimo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` il valore di `value` eccede `SEM_VALUE_MAX`.

`ENOSYS` il valore di `pshared` non è nullo ed il sistema non supporta i semafori per i processi.

La funzione inizializza un semaforo all'indirizzo puntato dall'argomento `sem`, e come per `sem_open` consente di impostare un valore iniziale con `value`. L'argomento `pshared` serve ad indicare se il semaforo deve essere utilizzato dai *thread* di uno stesso processo (con un valore nullo) o condiviso fra processi diversi (con un valore non nullo).

Qualora il semaforo debba essere condiviso dai *thread* di uno stesso processo (nel qual caso si parla di *thread-shared semaphore*), occorrerà che `sem` sia l'indirizzo di una variabile visibile da tutti i *thread*, si dovrà usare cioè una variabile globale o una variabile allocata dinamicamente nello *heap*.

Qualora il semaforo debba essere condiviso fra più processi (nel qual caso si parla di *process-shared semaphore*) la sola scelta possibile per renderlo visibile a tutti è di porlo in un tratto di memoria condivisa. Questo potrà essere ottenuto direttamente sia con `shmget` (vedi sez. 11.2.6) che con `shm_open` (vedi sez. 11.4.3), oppure, nel caso che tutti i processi in gioco abbiano un genitore comune, con una mappatura anonima con `mmap` (vedi sez. 10.4.1) a cui essi poi potranno accedere (si ricordi che i tratti di memoria condivisa vengono mantenuti nei processi figli attraverso la funzione `fork`).

Una volta inizializzato il semaforo anonimo con `sem_init` lo si potrà utilizzare nello stesso modo dei semafori normali con `sem_wait` e `sem_post`. Si tenga presente però che inizializzare due volte lo stesso semaforo può dar luogo ad un comportamento indefinito.

Qualora non si intenda più utilizzare un semaforo anonimo questo può essere eliminato dal sistema; per far questo si deve utilizzare una apposita funzione, `sem_destroy`, il cui prototipo è:

```
#include <semaphore.h>
int sem_destroy(sem_t *sem)
```

Elimina un semaforo anonimo.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EINVAL` l'argomento `sem` non indica un semaforo valido.

La funzione prende come unico argomento l'indirizzo di un semaforo che deve essere stato inizializzato con `sem_init`; non deve quindi essere applicata a semafori creati con `sem_open`. Inoltre si deve essere sicuri che il semaforo sia effettivamente inutilizzato, la distruzione di un semaforo su cui sono presenti processi (o *thread*) in attesa (cioè bloccati in una `sem_wait`) provoca un comportamento indefinito.

Si tenga presente infine che utilizzare un semaforo che è stato distrutto con `sem_destroy` di nuovo può dare esito a comportamenti indefiniti. Nel caso ci si trovi in una tale evenienza occorre reinizializzare il semaforo una seconda volta con `sem_init`.

Come esempio di uso sia della memoria condivisa che dei semafori POSIX si sono scritti due semplici programmi con i quali è possibile rispettivamente monitorare il contenuto di un segmento di memoria condivisa e modificarne il contenuto.

Il corpo principale del primo dei due, il cui codice completo è nel file `message_getter.c` dei sorgenti allegati, è riportato in fig. 11.32; si è tralasciata la parte che tratta la gestione delle opzioni a riga di comando (che consentono di impostare un nome diverso per il semaforo e il segmento di memoria condivisa) ed il controllo che al programma venga fornito almeno un argomento, contenente la stringa iniziale da inserire nel segmento di memoria condivisa.

Lo scopo del programma è quello di creare un segmento di memoria condivisa su cui registrare una stringa, e tenerlo sotto osservazione stampando la stessa una volta al secondo. Si utilizzerà un semaforo per proteggere l'accesso in lettura alla stringa, in modo che questa non possa essere modificata dall'altro programma prima di averla finita di stampare.

La parte iniziale del programma contiene le definizioni (1-8) del gestore del segnale usato per liberare le risorse utilizzate, delle variabili globali contenenti i nomi di default del segmento di memoria condivisa e del semaforo (il default scelto è `messages`), e delle altre variabili utilizzate dal programma.

Come prima istruzione (10) si è provveduto ad installare un gestore di segnale che consentirà di effettuare le operazioni di pulizia (usando la funzione `Signal` illustrata in fig. 7.10), dopo di che (12-16) si è creato il segmento di memoria condivisa con la funzione `CreateShm` che abbiamo appena trattato in sez. 11.4.3, uscendo con un messaggio in caso di errore.

Si tenga presente che la funzione `CreateShm` richiede che il segmento non sia già presente e fallirà qualora un'altra istanza, o un altro programma abbia già allocato un segmento con quello stesso nome. Per semplicità di gestione si è usata una dimensione fissa pari a 256 byte, definita tramite la costante `MSGMAXSIZE`.

Il passo successivo (17-21) è quello della creazione del semaforo che regola l'accesso al segmento di memoria condivisa con `sem_open`; anche in questo caso si gestisce l'uscita con stampa di un messaggio in caso di errore. Anche per il semaforo, avendo specificato la combinazione di flag `O_CREAT|O_EXCL` come secondo argomento, si esce qualora fosse già esistente; altrimenti esso verrà creato con gli opportuni permessi specificati dal terzo argomento, (indicante lettura e scrittura in notazione ottale). Infine il semaforo verrà inizializzato ad un valore nullo (il quarto argomento), corrispondente allo stato in cui risulta bloccato.

A questo punto (22) si potrà inizializzare il messaggio posto nel segmento di memoria condivisa usando la stringa passata come argomento al programma. Essendo il semaforo stato creato già bloccato non ci si dovrà preoccupare di eventuali *race condition* qualora il programma di modifica del messaggio venisse lanciato proprio in questo momento. Una volta inizializzato il messaggio occorrerà però rilasciare il semaforo (24-27) per consentirne l'uso; in tutte queste operazioni si provvederà ad uscire dal programma con un opportuno messaggio in caso di errore.

Una volta completate le inizializzazioni il ciclo principale del programma (29-47) viene ripetuto indefinitamente (29) per stampare sia il contenuto del messaggio che una serie di informazioni di controllo. Il primo passo (30-34) è quello di acquisire (con `sem_getvalue`, con uscita in caso di errore) e stampare il valore del semaforo ad inizio del ciclo; seguito (35-36) dal tempo corrente.

Prima della stampa del messaggio invece si deve acquisire il semaforo (30-33) per evitare accessi concorrenti alla stringa da parte del programma di modifica. Una volta eseguita la

```

1 void HandSigInt(int sig);
2 #define MSGMAXSIZE 256
3 char *shmname = "messages";
4 char *semname = "messages";
5
6 int main(int argc, char *argv[])
7 {
8     sem_t * sem, void * shm_ptr, time_t t;
9     ...
10    Signal(SIGINT, HandSigInt);
11    // get a shared memory segment
12    if ((shm_ptr = CreateShm(shmname, MSGMAXSIZE, 0666, 0)) == NULL) {
13        perror("Cannot_find_shared_memory");
14        exit(1);
15    }
16    // get a locked semaphore
17    if ((sem = sem_open(semname, O_CREAT|O_EXCL, 0666, 0)) == SEM_FAILED) {
18        perror("Cannot_open_semaphore");
19        exit(1);
20    }
21    // set initial string
22    strncpy((char *) shm_ptr, argv[optind], MSGMAXSIZE);
23    // do initial release
24    if (sem_post(sem) != 0) {
25        perror("cannot_do_semaphore_initial_release");
26        exit(1);
27    }
28    // main loop
29    while(1) {
30        if (sem_getvalue(sem, &i) !=0) {                // get sem values
31            perror("cannot_get_semaphore_value");
32            exit(1);
33        }
34        printf("sem=%i,\n", i);                          // print sem values
35        t = time(NULL);                                  // get time
36        printf("%s", ctime(&t));                          // print time
37        if (sem_wait(sem) != 0) {                        // acquire semaphore
38            perror("cannot_use_semaphore");
39            exit(1);
40        }
41        printf("message:_%s\n", (char *) shm_ptr );      // print message
42        if (sem_post(sem) != 0) {                        // release semaphore
43            perror("cannot_release_semaphore");
44            exit(1);
45        }
46        sleep(1);
47    }
48    exit(0);
49 }

```

Figura 11.32: Sezione principale del codice del programma `message_getter.c`.

```

1 void HandSigInt(int sig)
2 {
3     if (RemoveShm(shmname) != 0) perror("Cannot_remove_shared_memory");
4     if (sem_unlink(semname) != 0) perror("Cannot_remove_semaphore");
5     exit(0);
6 }

```

Figura 11.33: Codice del gestore di segnale del programma `message_getter.c`.

stampa (41) il semaforo dovrà essere rilasciato (42-45). Il passo finale (46) è attendere per un secondo prima di eseguire da capo il ciclo.

Per uscire in maniera corretta dal programma sarà necessario fermarlo con una interruzione da tastiera (C-c), che corrisponde all'invio del segnale `SIGINT`, per il quale si è installato (10) una opportuna funzione di gestione, riportata in fig. 11.33. La funzione è molto semplice e richiama le funzioni di rimozione sia per il segmento di memoria condivisa che per il semaforo, garantendo così che possa essere riaperto ex-novo senza errori in un futuro riutilizzo del comando.

Il secondo programma di esempio è `message_setter.c`, di cui si è riportato il corpo principale in fig. 11.34,²⁹ dove si è tralasciata, non essendo significativa per quanto si sta trattando, la parte relativa alla gestione delle opzioni a riga di comando e degli argomenti, che sono identici a quelli usati da `message_getter`, con l'unica aggiunta di un'opzione `"-t"` che consente di indicare un tempo di attesa (in secondi) in cui il programma si ferma tenendo bloccato il semaforo.

Una volta completata la gestione delle opzioni e degli argomenti (ne deve essere presente uno solo, contenente la nuova stringa da usare come messaggio), il programma procede (10-14) con l'acquisizione del segmento di memoria condivisa usando la funzione `FindShm` (trattata in sez. 11.4.3) che stavolta deve già esistere. Il passo successivo (16-19) è quello di aprire il semaforo, e a differenza di `message_getter`, in questo caso si richiede a `sem_open` che questo esista, passando uno zero come secondo ed unico argomento.

Una volta completate con successo le precedenti inizializzazioni, il passo seguente (21-24) è quello di acquisire il semaforo, dopo di che sarà possibile eseguire la sostituzione del messaggio (25) senza incorrere in possibili *race condition* con la stampa dello stesso da parte di `message_getter`.

Una volta effettuata la modifica viene stampato (26) il tempo di attesa impostato con l'opzione `"-t"` dopo di che (27) viene eseguita la stessa, senza rilasciare il semaforo che resterà quindi bloccato (causando a questo punto una interruzione delle stampe eseguite da `message_getter`). Terminato il tempo di attesa si rilascerà (29-32) il semaforo per poi uscire.

Per verificare il funzionamento dei programmi occorrerà lanciare prima `message_getter` (lanciare per primo `message_setter` darebbe luogo ad un errore, non essendo stati creati il semaforo ed il segmento di memoria condivisa) che inizierà a stampare una volta al secondo il contenuto del messaggio ed i suoi dati, con qualcosa del tipo:

```

piccardi@hain:~/gabil/sources$ ./message_getter messaggio
sem=1, Fri Dec 31 14:12:41 2010
message: messaggio
sem=1, Fri Dec 31 14:12:42 2010
message: messaggio
...

```

²⁹al solito il codice completo è nel file dei sorgenti allegati.

```

1 #define MSGMAXSIZE 256
2
3 int main(int argc, char *argv[])
4 {
5     int t = 0, sem_t * sem, void *shm_ptr;
6     char *shmname = "messages";
7     char *semname = "messages";
8     ...
9     // get shared memory segment
10    shm_ptr = FindShm(shmname, MSGMAXSIZE);
11    if ( shm_ptr == NULL ) {
12        perror("Cannot_find_shared_memory");
13        exit(1);
14    }
15    // open semaphore
16    if ( (sem = sem_open(semname, 0)) == SEM_FAILED ) {
17        perror("Cannot_open_semaphore");
18        exit(1);
19    }
20    // get semaphore
21    if ( sem_wait(sem) != 0 ) {
22        perror("cannot_use_semaphore");
23        exit(1);
24    }
25    strncpy((char *) shm_ptr, argv[optind], MSGMAXSIZE); // modify message
26    printf("Sleeping_for_%i_seconds\n", t); // print wait time
27    sleep(t); // sleep
28    // release semaphore
29    if ( sem_post(sem) != 0 ) {
30        perror("cannot_release_semaphore");
31        exit(1);
32    }
33    exit(0);
34 }

```

Figura 11.34: Sezione principale del codice del programma `message_setter.c`.

proseguendo indefinitamente fintanto che non si preme C-c per farlo uscire. Si noti come il valore del semaforo risulti sempre pari ad 1 (in quanto al momento esso sarà sempre libero).

A questo punto si potrà lanciare `message_setter` per cambiare il messaggio, nel nostro caso per rendere evidente il funzionamento del blocco richiederemo anche una attesa di 3 secondi, ed otterremo qualcosa del tipo:

```

piccardi@hain:~/gapil/sources$ ./message_setter -t 3 ciao
Sleeping for 3 seconds

```

dove il programma si fermerà per 3 secondi prima di rilasciare il semaforo e terminare. L'effetto di tutto ciò si potrà vedere nell'*output* di `message_getter`, che verrà interrotto per questo stesso tempo, prima di ricominciare con il nuovo testo:

```

...
sem=1, Fri Dec 31 14:16:27 2010
message: messaggio
sem=1, Fri Dec 31 14:16:28 2010
message: messaggio
sem=0, Fri Dec 31 14:16:29 2010
message: ciao

```

```
sem=1, Fri Dec 31 14:16:32 2010  
message: ciao  
sem=1, Fri Dec 31 14:16:33 2010  
message: ciao  
...
```

E si noterà come nel momento in cui si lancia `message_setter` le stampe di `message_getter` si bloccheranno, come corretto, dopo aver registrato un valore nullo per il semaforo. Il programma infatti resterà bloccato nella `sem_wait` (quella di riga (37) in fig. 11.32) fino alla scadenza dell'attesa di `message_setter` (con l'esecuzione della `sem_post` della riga (29) di fig. 11.34), e riprenderanno con il nuovo testo alla terminazione di quest'ultimo.

Capitolo 12

I thread

Tratteremo in questo capitolo un modello di programmazione multitasking, quello dei *thread*, alternativo al modello classico dei processi, tipico di Unix. Ne esamineremo le caratteristiche, vantaggi e svantaggi, e le diverse realizzazioni che sono disponibili per Linux; nella seconda parte tratteremo in dettaglio quella che è l'implementazione principale, che fa riferimento all'interfaccia standardizzata da POSIX.1e.

12.1 Introduzione ai *thread*

Questa prima sezione costituisce una introduzione ai *thread* e tratterà i concetti principali del relativo modello di programmazione, esamineremo anche quali modelli sono disponibili per Linux, dando una breve panoramica sulle implementazioni alternative.

12.1.1 Una panoramica

Il modello classico dell'esecuzione dei programmi nei sistemi Unix, illustrato in sez. 2, è fondato sui processi. Il modello nasce per assicurare la massima stabilità al sistema e prevede una rigida separazione fra i diversi processi, in modo che questi non possano disturbarsi a vicenda.

Le applicazioni moderne però sono altamente concorrenti, e necessitano quindi di un gran numero di processi; questo ha portato a scontrarsi con alcuni limiti dell'architettura precedente. In genere i fautori del modello di programmazione a *thread* sottolineano due problemi connessi all'uso dei processi:

-
-

12.1.2 *Thread* e processi

Per un utilizzo effettivo dei *thread* è sempre opportuno capire se questi sono davvero adatti allo scopo che ci si pone.

12.1.3 Implementazioni alternative

Vedremo nella prossima sezione le caratteristiche del supporto per i *thread* fornita dal kernel, ma esistono diversi possibili approcci alle modalità in cui questi possono essere realizzati.

12.2 I *thread* e Linux

In questa sezione tratteremo le implementazioni dei *thread* disponibili con Linux che ha visto un radicale cambiamento nel passaggio dalla serie 2.4 alla serie 2.6, che ha portato alla versione attuale.

12.2.1 I *LinuxThread*

12.2.2 La *Native Thread Posix Library*

12.3 Posix *thread*

Tratteremo in questa sezione l'interfaccia di programmazione con i *thread* standardizzata dallo standard POSIX 1.c, che è quella che è stata seguita anche dalle varie implementazioni dei *thread* realizzate su Linux, ed in particolare dalla *Native Thread Posix Library* che è stata integrata con i kernel della serie 2.6 e che fa parte a pieno titolo della *glibc*.

12.3.1 Una panoramica

12.3.2 La gestione dei *thread*

Benché la funzione sia utilizzabile anche con i processi, tanto che a partire dalla versione 2.3 della *glibc* viene a sostituire `_exit` (tramite un *wrapper* che la utilizza al suo posto) per la terminazione di tutti i *thread* di un processo si deve usare la funzione di sistema `exit_group`, il cui prototipo è:

```
#include <linux/unistd.h>
void exit_group(int status)
```

Termina tutti i *thread* di un processo.

La funzione non ha errori e pertanto non ritorna.

La funzione è sostanzialmente identica alla *system call* `_exit` ma a differenza di quest'ultima, che termina solo il *thread* chiamante, termina tutti *thread* del processo.

12.4 La sincronizzazione dei *thread*

12.4.1 I *mutex*

12.4.2 Le variabili di condizione

12.4.3 I *thread* e i segnali.

Parte II

Programmazione di rete

Capitolo 13

Introduzione alla programmazione di rete

In questo capitolo sarà fatta un'introduzione ai concetti generali che servono come prerequisiti per capire la programmazione di rete, non tratteremo quindi aspetti specifici ma faremo una breve introduzione ai modelli più comuni usati nella programmazione di rete, per poi passare ad un esame a grandi linee dei protocolli di rete e di come questi sono organizzati e interagiscono.

In particolare, avendo assunto l'ottica di un'introduzione mirata alla programmazione, ci concentreremo sul gruppo di protocolli più diffuso, il TCP/IP, che è quello che sta alla base di Internet, avendo cura di sottolineare i concetti più importanti da conoscere per la scrittura dei programmi.

13.1 Modelli di programmazione

La differenza principale fra un'applicazione di rete e un programma normale è che quest'ultima per definizione concerne la comunicazione fra processi diversi, che in generale non girano neanche sulla stessa macchina. Questo già prefigura un cambiamento completo rispetto all'ottica del programma monolitico all'interno del quale vengono eseguite tutte le istruzioni, e chiaramente presuppone un sistema operativo multitasking in grado di eseguire più processi contemporaneamente.

In questa prima sezione esamineremo brevemente i principali modelli di programmazione in uso. Ne daremo una descrizione assolutamente generica e superficiale, che ne illustri le caratteristiche principali, non essendo fra gli scopi del testo approfondire questi argomenti.

13.1.1 Il modello *client-server*

L'architettura fondamentale su cui si basa gran parte della programmazione di rete sotto Linux (e sotto Unix in generale) è il modello *client-server* caratterizzato dalla presenza di due categorie di soggetti, i programmi di servizio, chiamati *server*, che ricevono le richieste e forniscono le risposte, ed i programmi di utilizzo, detti *client*.

In generale un server può (di norma deve) essere in grado di rispondere a più di un client, per cui è possibile che molti programmi possano interagire contemporaneamente, quello che

contraddistingue il modello però è che l'architettura dell'interazione è sempre nei termini di molti verso uno, il server, che viene ad assumere un ruolo privilegiato.

Seguono questo modello tutti i servizi fondamentali di Internet, come le pagine web, la posta elettronica, ftp, telnet, ssh e praticamente ogni servizio che viene fornito tramite la rete, anche se, come abbiamo visto, il modello è utilizzato in generale anche per programmi che non fanno necessariamente uso della rete, come gli esempi che abbiamo usato in cap. 11 a proposito della comunicazione fra processi nello stesso sistema.

Normalmente si dividono i server in due categorie principali, e vengono detti *concorrenti* o *iterativi*, sulla base del loro comportamento. Un *server iterativo* risponde alla richiesta inviando i dati e resta occupato e non rispondendo ad ulteriori richieste fintanto che non ha fornito una risposta alla richiesta. Una volta completata la risposta il server diventa di nuovo disponibile.

Un *server concorrente* al momento di trattare la richiesta crea un processo figlio (o un *thread*) incaricato di fornire i servizi richiesti, per porsi immediatamente in attesa di ulteriori richieste. In questo modo, con sistemi multitasking, più richieste possono essere soddisfatte contemporaneamente. Una volta che il processo figlio ha concluso il suo lavoro esso di norma viene terminato, mentre il server originale resta sempre attivo.

13.1.2 Il modello *peer-to-peer*

Come abbiamo visto il tratto saliente dell'architettura *client-server* è quello della preminenza del server rispetto ai client, le architetture *peer-to-peer* si basano su un approccio completamente opposto che è quello di non avere nessun programma che svolga un ruolo preminente.

Questo vuol dire che in generale ciascun programma viene ad agire come un nodo in una rete potenzialmente paritetica; ciascun programma si trova pertanto a ricevere ed inviare richieste ed a ricevere ed inviare risposte, e non c'è più la separazione netta dei compiti che si ritrova nelle architetture *client-server*.

Le architetture *peer-to-peer* sono salite alla ribalta con l'esplosione del fenomeno Napster, ma gli stessi protocolli di routing sono un buon esempio di architetture *peer-to-peer*, in cui ciascun nodo, tramite il demone che gestisce il routing, richiede ed invia informazioni ad altri nodi.

In realtà in molti casi di architetture classificate come *peer-to-peer* non è detto che la struttura sia totalmente paritetica e ci sono parecchi esempi in cui alcuni servizi vengono centralizzati o distribuiti gerarchicamente, come avveniva per lo stesso Napster, in cui le ricerche erano effettuate su un server centrale.

13.1.3 Il modello *three-tier*

Benché qui sia trattato a parte, il modello *three-tier* in realtà è una estensione del modello *client-server*. Con il crescere della quantità dei servizi forniti in rete (in particolare su Internet) ed al numero di accessi richiesto. Si è così assistito anche ad una notevole crescita di complessità, in cui diversi servizi venivano ad essere integrati fra di loro.

In particolare sempre più spesso si assiste ad una integrazione di servizi di database con servizi di web, in cui le pagine vengono costruite dinamicamente sulla base dei dati contenuti nel database. In tutti questi casi il problema fondamentale di una architettura *client-server* è che la richiesta di un servizio da parte di un gran numero di client si scontra con il collo di bottiglia dell'accesso diretto ad un unico server, con gravi problemi di scalabilità.

Rispondere a queste esigenze di scalabilità il modello più semplice (chiamato talvolta *two-tier*) da adottare è stata quello di distribuire il carico delle richieste su più server identici, mantenendo quindi sostanzialmente inalterata l'architettura *client-server* originale.

Nel far questo ci si scontra però con gravi problemi di manutenibilità dei servizi, in particolare per quanto riguarda la sincronizzazione dei dati, e di inefficienza dell'uso delle risorse. Il problema è particolarmente grave ad esempio per i database che non possono essere replicati e sincronizzati facilmente, e che sono molto onerosi, la loro replicazione è costosa e complessa.

È a partire da queste problematiche che nasce il modello *three-tier*, che si struttura, come dice il nome, su tre livelli. Il primo livello, quello dei client che eseguono le richieste e gestiscono l'interfaccia con l'utente, resta sostanzialmente lo stesso del modello *client-server*, ma la parte server viene suddivisa in due livelli, introducendo un *middle-tier*, su cui deve appoggiarsi tutta la logica di analisi delle richieste dei client per ottimizzare l'accesso al terzo livello, che è quello che si limita a fornire i dati dinamici che verranno usati dalla logica implementata nel *middle-tier* per eseguire le operazioni richieste dai client.

In questo modo si può disaccoppiare la logica dai dati, replicando la prima, che è molto meno soggetta a cambiamenti ed evoluzione, e non soffre di problemi di sincronizzazione, e centralizzando opportunamente i secondi. In questo modo si può distribuire il carico ed accedere in maniera efficiente i dati.

13.1.4 Il modello *broadcast*

Uno specifico modello relativo alla programmazione di rete è poi quello in cui è possibile, invece della classica comunicazione uno ad uno comunque usata in tutti i modelli precedenti (anche nel *peer-to-peer* la comunicazione è comunque fra singoli "*peer*"), una comunicazione da uno a molti.

Questo modello nasce dal fatto che molte tecnologie di rete (ed in particolare Ethernet, che è probabilmente la più diffusa) hanno il supporto per effettuare una comunicazione in cui un nodo qualunque della rete può inviare informazioni in contemporanea a tutti gli altri. In questo caso si parla di *broadcast*, utilizzando la nomenclatura usata per le trasmissioni radio, anche se in realtà questo tipo di comunicazione è eseguibile da un nodo qualunque per cui tutti quanti possono ricoprire sia il ruolo di trasmettitore che quello di ricevitore.

In genere si parla di *broadcast* quando la trasmissione uno a molti è possibile fra qualunque nodo di una rete e gli altri, ed è supportata direttamente dalla tecnologia di collegamento utilizzata. L'utilizzo di questa forma di comunicazione da uno a molti però può risultare molto utile anche quando questo tipo di supporto non è disponibile (come ad esempio su Internet, dove non si possono contattare tutti i nodi presenti).

In tal caso alcuni protocolli di rete (e quelli usati per Internet sono fra questi) supportano una variante del *broadcast*, detta *multicast*, in cui resta possibile fare una comunicazione uno a molti, in cui una applicazione invia i pacchetti a molte altre, in genere passando attraverso un opportuno supporto degli apparati ed una qualche forma di registrazione che consente la distribuzione della comunicazione ai nodi interessati.

Ovviamente i programmi che devono realizzare un tipo di comunicazione di questo tipo (come ad esempio potrebbero essere quelli che effettuano uno *streaming* di informazioni) devono rispondere a delle problematiche del tutto diverse da quelle classiche illustrate nei modelli precedenti, e costituiscono pertanto un'altra classe completamente a parte.

13.2 I protocolli di rete

Parlando di reti di computer si parla in genere di un insieme molto vasto ed eterogeneo di mezzi di comunicazione che vanno dal cavo telefonico, alla fibra ottica, alle comunicazioni via satellite o via radio; per rendere possibile la comunicazione attraverso un così variegato insieme di mezzi sono stati adottati molti protocolli, il più famoso dei quali, quello alla base del funzionamento di Internet, è il gruppo di protocolli comunemente chiamato TCP/IP.

13.2.1 Il modello ISO/OSI

Una caratteristica comune dei protocolli di rete è il loro essere strutturati in livelli sovrapposti; in questo modo ogni protocollo di un certo livello realizza le sue funzionalità basandosi su un protocollo del livello sottostante. Questo modello di funzionamento è stato standardizzato dalla *International Standards Organization* (ISO) che ha preparato fin dal 1984 il Modello di Riferimento *Open Systems Interconnection* (OSI), strutturato in sette livelli, secondo quanto riportato in tab. 13.1.

Livello	Nome	
Livello 7	<i>Application</i>	<i>Applicazione</i>
Livello 6	<i>Presentation</i>	<i>Presentazione</i>
Livello 5	<i>Session</i>	<i>Sessione</i>
Livello 4	<i>Transport</i>	<i>Trasporto</i>
Livello 3	<i>Network</i>	<i>Rete</i>
Livello 2	<i>DataLink</i>	<i>Collegamento Dati</i>
Livello 1	<i>Physical</i>	<i>Connessione Fisica</i>

Tabella 13.1: I sette livelli del protocollo ISO/OSI.

Il modello ISO/OSI è stato sviluppato in corrispondenza alla definizione della serie di protocolli X.25 per la commutazione di pacchetto; come si vede è un modello abbastanza complesso¹, tanto che usualmente si tende a suddividerlo in due parti, secondo lo schema mostrato in fig. 13.1, con un *upper layer* che riguarda solo le applicazioni, che viene realizzato in *user space*, ed un *lower layer* in cui si mescolano la gestione fatta dal kernel e le funzionalità fornite dall'hardware.

Il modello ISO/OSI mira ad effettuare una classificazione completamente generale di ogni tipo di protocollo di rete; nel frattempo però era stato sviluppato anche un altro modello, relativo al protocollo TCP/IP, che è quello su cui è basata Internet, che è diventato uno standard de facto. Questo modello viene talvolta chiamato anche modello *DoD* (sigla che sta per *Department of Defense*), dato che fu sviluppato dall'agenzia ARPA per il Dipartimento della Difesa Americano.

Figura 13.1: Struttura a livelli dei protocolli OSI e TCP/IP, con la relative corrispondenze e la divisione fra *kernel space* e *user space*.

La scelta fra quale dei due modelli utilizzare dipende per lo più dai gusti personali. Come caratteristiche generali il modello ISO/OSI è più teorico e generico, basato separazioni funzionali, mentre il modello TCP/IP è più vicino alla separazione concreta dei vari strati

¹infatti per memorizzarne i vari livelli è stata creata la frase *All people seem to need data processing*, in cui ciascuna parola corrisponde all'iniziale di uno dei livelli.

del sistema operativo; useremo pertanto quest'ultimo, anche per la sua maggiore semplicità. Questa semplicità ha un costo quando si fa riferimento agli strati più bassi, che sono in effetti descritti meglio dal modello ISO/OSI, in quanto gran parte dei protocolli di trasmissione hardware sono appunto strutturati sui due livelli di *Data Link* e *Connection*.

13.2.2 Il modello TCP/IP (o DoD)

Così come ISO/OSI anche il modello del TCP/IP è stato strutturato in livelli (riassunti in tab. 13.2); un confronto fra i due è riportato in fig. 13.1 dove viene evidenziata anche la corrispondenza fra i rispettivi livelli (che comunque è approssimativa) e su come essi vanno ad inserirsi all'interno del sistema rispetto alla divisione fra *user space* e *kernel space* spiegata in sez. 1.1.²

Livello	Nome		Esempi
Livello 4	<i>Application</i>	Applicazione	Telnet, FTP, ecc.
Livello 3	<i>Transport</i>	Trasporto	TCP, UDP
Livello 2	<i>Network</i>	Rete	IP, (ICMP, IGMP)
Livello 1	<i>Link</i>	Collegamento	Device driver & scheda di interfaccia

Tabella 13.2: I quattro livelli del protocollo TCP/IP.

Come si può notare come il modello TCP/IP è più semplice del modello ISO/OSI ed è strutturato in soli quattro livelli. Il suo nome deriva dai due principali protocolli che lo compongono, il TCP (*Transmission Control Protocol*) che copre il livello 3 e l'IP (*Internet Protocol*) che copre il livello 2. Le funzioni dei vari livelli sono le seguenti:

Applicazione	É relativo ai programmi di interfaccia con la rete, in genere questi vengono realizzati secondo il modello client-server (vedi sez. 13.1.1), realizzando una comunicazione secondo un protocollo che è specifico di ciascuna applicazione.
Trasporto	Fornisce la comunicazione tra le due stazioni terminali su cui girano gli applicativi, regola il flusso delle informazioni, può fornire un trasporto affidabile, cioè con recupero degli errori o inaffidabile. I protocolli principali di questo livello sono il TCP e l'UDP.
Rete	Si occupa dello smistamento dei singoli pacchetti su una rete complessa e interconnessa, a questo stesso livello operano i protocolli per il reperimento delle informazioni necessarie allo smistamento, per lo scambio di messaggi di controllo e per il monitoraggio della rete. Il protocollo su cui si basa questo livello è IP (sia nella attuale versione, IPv4, che nella nuova versione, IPv6).
Collegamento	È responsabile per l'interfacciamento al dispositivo elettronico che effettua la comunicazione fisica, gestendo l'invio e la ricezione dei pacchetti da e verso l'hardware.

La comunicazione fra due stazioni remote avviene secondo le modalità illustrate in fig. 13.2, dove si è riportato il flusso dei dati reali e i protocolli usati per lo scambio di informazione su ciascun livello. Si è genericamente indicato *ethernet* per il livello 1, anche se in realtà i protocolli di trasmissione usati possono essere molti altri.

²in realtà è sempre possibile accedere dallo *user space*, attraverso una opportuna interfaccia (come vedremo in sez. 14.2.6), ai livelli inferiori del protocollo.

Figura 13.2: Strutturazione del flusso dei dati nella comunicazione fra due applicazioni attraverso i protocolli della suite TCP/IP.

Per chiarire meglio la struttura della comunicazione attraverso i vari protocolli mostrata in fig. 13.2, conviene prendere in esame i singoli passaggi fatti per passare da un livello al sottostante, la procedura si può riassumere nei seguenti passi:

- Le singole applicazioni comunicano scambiandosi i dati ciascuna secondo un suo specifico formato. Per applicazioni generiche, come la posta o le pagine web, viene di solito definito ed implementato quello che viene chiamato un protocollo di applicazione (esempi possono essere HTTP, POP, SMTP, ecc.), ciascuno dei quali è descritto in un opportuno standard, di solito attraverso un RFC (l'acronimo RFC sta per *Request For Comment* ed è la procedura attraverso la quale vengono proposti gli standard per Internet).
- I dati delle applicazioni vengono inviati al livello di trasporto usando un'interfaccia opportuna (i *socket*, che esamineremo in dettaglio in cap. 14). Qui verranno spezzati in pacchetti di dimensione opportuna e inseriti nel protocollo di trasporto, aggiungendo ad ogni pacchetto le informazioni necessarie per la sua gestione. Questo processo viene svolto direttamente nel kernel, ad esempio dallo stack TCP, nel caso il protocollo di trasporto usato sia questo.
- Una volta composto il pacchetto nel formato adatto al protocollo di trasporto usato questo sarà passato al successivo livello, quello di rete, che si occupa di inserire le opportune informazioni per poter effettuare l'instradamento nella rete ed il recapito alla destinazione finale. In genere questo è il livello di IP (Internet Protocol), a cui vengono inseriti i numeri IP che identificano i computer su Internet.
- L'ultimo passo è il trasferimento del pacchetto al driver della interfaccia di trasmissione, che si incarica di incapsularlo nel relativo protocollo di trasmissione. Questo può avvenire sia in maniera diretta, come nel caso di ethernet, in cui i pacchetti vengono inviati sulla linea attraverso le schede di rete, che in maniera indiretta con protocolli come PPP o SLIP, che vengono usati come interfaccia per far passare i dati su altri dispositivi di comunicazione (come la seriale o la parallela).

13.2.3 Criteri generali dell'architettura del TCP/IP

La filosofia architetturale del TCP/IP è semplice: costruire una rete che possa sopportare il carico in transito, ma permettere ai singoli nodi di scartare pacchetti se il carico è temporaneamente eccessivo, o se risultano errati o non recapitabili.

L'incarico di rendere il recapito pacchetti affidabile non spetta al livello di rete, ma ai livelli superiori. Pertanto il protocollo IP è per sua natura inaffidabile, in quanto non è assicurata né una percentuale di successo né un limite sui tempi di consegna dei pacchetti.

È il livello di trasporto che si deve occupare (qualora necessari) del controllo del flusso dei dati e del recupero degli errori; questo è realizzato dal protocollo TCP. La sede principale di "intelligenza" della rete è pertanto al livello di trasporto o ai livelli superiori.

Infine le singole stazioni collegate alla rete non fungono soltanto da punti terminali di comunicazione, ma possono anche assumere il ruolo di *router* (*instradatori*), per l'interscambio di pacchetti da una rete ad un'altra. Questo rende possibile la flessibilità della rete che è in grado di adattarsi ai mutamenti delle interconnessioni.

La caratteristica essenziale che rende tutto ciò possibile è la strutturazione a livelli tramite l'incapsulamento. Ogni pacchetto di dati viene incapsulato nel formato del livello successivo, fino al livello del collegamento fisico. In questo modo il pacchetto ricevuto ad un livello n dalla stazione di destinazione è esattamente lo stesso spedito dal livello n dalla sorgente. Questo rende facile il progettare il software facendo riferimento unicamente a quanto necessario ad un singolo livello, con la confidenza che questo poi sarà trattato uniformemente da tutti i nodi della rete.

13.3 La struttura del TCP/IP

Come accennato in sez. 13.2 il TCP/IP è un insieme di protocolli diversi, che operano su 4 livelli diversi. Per gli interessi della programmazione di rete però sono importanti principalmente i due livelli centrali, e soprattutto quello di trasporto.

La principale interfaccia usata nella programmazione di rete, quella dei socket (che vedremo in sez. 14), è infatti un'interfaccia nei confronti di quest'ultimo. Questo avviene perché al di sopra del livello di trasporto i programmi hanno a che fare solo con dettagli specifici delle applicazioni, mentre al di sotto vengono curati tutti i dettagli relativi alla comunicazione. È pertanto naturale definire una interfaccia di programmazione su questo confine, tanto più che è proprio lì (come evidenziato in fig. 13.1) che nei sistemi Unix (e non solo) viene inserita la divisione fra *kernel space* e *user space*.

In realtà in un sistema Unix è possibile accedere anche agli altri livelli (e non solo a quello di trasporto) con opportune interfacce di programmazione (vedi sez. 14.2.6), ma queste vengono usate solo quando si debbano fare applicazioni di sistema per il controllo della rete a basso livello, di uso quindi molto specialistico.

In questa sezione daremo una descrizione sommaria dei vari protocolli del TCP/IP, concentrandoci, per le ragioni appena esposte, sul livello di trasporto. All'interno di quest'ultimo privilegeremo poi il protocollo TCP, per il ruolo centrale che svolge nella maggior parte delle applicazioni.

13.3.1 Il quadro generale

Benché si parli di TCP/IP questa famiglia di protocolli è composta anche da molti membri. In fig. 13.3 si è riportato uno schema che mostra un panorama sui principali protocolli della famiglia, e delle loro relazioni reciproche e con alcune delle principali applicazioni che li usano.

Figura 13.3: Panoramica sui vari protocolli che compongono la suite TCP/IP.

I vari protocolli riportati in fig. 13.3 sono i seguenti:

- | | |
|-------------|---|
| IPv4 | <i>Internet Protocol version 4.</i> È quello che comunemente si chiama IP. Ha origine negli anni '80 e da allora è la base su cui è costruita Internet. Usa indirizzi a 32 bit, e mantiene tutte le informazioni di instradamento e controllo per la trasmissione dei pacchetti sulla rete; tutti gli altri protocolli della suite (eccetto ARP e RARP, e quelli specifici di IPv6) vengono trasmessi attraverso di esso. |
| IPv6 | <i>Internet Protocol version 6.</i> È stato progettato a metà degli anni '90 per rimpiazzare IPv4. Ha uno spazio di indirizzi ampliato 128 bit che consente più gerarchie di indirizzi, l'auto-configurazione, ed un nuovo tipo di indirizzi, gli <i>anycast</i> , che |

consentono di inviare un pacchetto ad una stazione su un certo gruppo. Effettua lo stesso servizio di trasmissione dei pacchetti di IPv4 di cui vuole essere un sostituto.

- TCP** *Transmission Control Protocol.* È un protocollo orientato alla connessione che provvede un trasporto affidabile per un flusso di dati bidirezionale fra due stazioni remote. Il protocollo ha cura di tutti gli aspetti del trasporto dei dati, come l'*acknowledgment* (il ricevuto), i timeout, la ritrasmissione, ecc. È usato dalla maggior parte delle applicazioni.
- UDP** *User Datagram Protocol.* È un protocollo senza connessione, per l'invio di dati a pacchetti. Contrariamente al TCP il protocollo non è affidabile e non c'è garanzia che i pacchetti raggiungano la loro destinazione, si perdano, vengano duplicati, o abbiano un particolare ordine di arrivo.
- ICMP** *Internet Control Message Protocol.* È il protocollo usato a livello 2 per gestire gli errori e trasportare le informazioni di controllo fra stazioni remote e instradatori (cioè fra *host* e *router*). I messaggi sono normalmente generati dal software del kernel che gestisce la comunicazione TCP/IP, anche se ICMP può venire usato direttamente da alcuni programmi come ping. A volte ci si riferisce ad esso come ICMPv4 per distinguerlo da ICMPv6.
- IGMP** *Internet Group Management Protocol.* È un protocollo di livello 2 usato per il *multicast* (vedi sez. ??). Permette alle stazioni remote di notificare ai router che supportano questa comunicazione a quale gruppo esse appartengono. Come ICMP viene implementato direttamente sopra IP.
- ARP** *Address Resolution Protocol.* È il protocollo che mappa un indirizzo IP in un indirizzo hardware sulla rete locale. È usato in reti di tipo *broadcast* come Ethernet, Token Ring o FDDI che hanno associato un indirizzo fisico (il *MAC address*) alla interfaccia, ma non serve in connessioni punto-punto.
- RARP** *Reverse Address Resolution Protocol.* È il protocollo che esegue l'operazione inversa rispetto ad ARP (da cui il nome) mappando un indirizzo hardware in un indirizzo IP. Viene usato a volte per durante l'avvio per assegnare un indirizzo IP ad una macchina.
- ICMPv6** *Internet Control Message Protocol, version 6.* Combina per IPv6 le funzionalità di ICMPv4, IGMP e ARP.
- EGP** *Exterior Gateway Protocol.* È un protocollo di routing usato per comunicare lo stato fra gateway vicini a livello di *sistemi autonomi* (vengono chiamati *autonomous systems* i raggruppamenti al livello più alto della rete), con meccanismi che permettono di identificare i vicini, controllarne la raggiungibilità e scambiare informazioni sullo stato della rete. Viene implementato direttamente sopra IP.
- OSPF** *Open Shortest Path First.* È in protocollo di routing per router su reti interne, che permette a questi ultimi di scambiarsi informazioni sullo stato delle connessioni e dei legami che ciascuno ha con gli altri. Viene implementato direttamente sopra IP.
- GRE** *Generic Routing Encapsulation.* È un protocollo generico di incapsulamento che permette di incapsulare un qualunque altro protocollo all'interno di IP.

<i>AH</i>	<i>Authentication Header</i> . Provvede l'autenticazione dell'integrità e dell'origine di un pacchetto. È una opzione nativa in IPv6 e viene implementato come protocollo a sé su IPv4. Fa parte della suite di IPSEC che provvede la trasmissione cifrata ed autenticata a livello IP.
<i>ESP</i>	<i>Encapsulating Security Payload</i> . Provvede la cifratura insieme all'autenticazione dell'integrità e dell'origine di un pacchetto. Come per AH è opzione nativa in IPv6 e viene implementato come protocollo a sé su IPv4.
<i>PPP</i>	<i>Point-to-Point Protocol</i> . È un protocollo a livello 1 progettato per lo scambio di pacchetti su connessioni punto punto. Viene usato per configurare i collegamenti, definire i protocolli di rete usati ed incapsulare i pacchetti di dati. È un protocollo complesso con varie componenti.
<i>SLIP</i>	<i>Serial Line over IP</i> . È un protocollo di livello 1 che permette di trasmettere un pacchetto IP attraverso una linea seriale.

Gran parte delle applicazioni comunicano usando TCP o UDP, solo alcune, e per scopi particolari si rifanno direttamente ad IP (ed i suoi correlati ICMP e IGMP); benché sia TCP che UDP siano basati su IP e sia possibile intervenire a questo livello con i *raw socket* questa tecnica è molto meno diffusa e a parte applicazioni particolari si preferisce sempre usare i servizi messi a disposizione dai due protocolli precedenti. Per questo, motivo a parte alcuni brevi accenni su IP in questa sezione, ci concentreremo sul livello di trasporto.

13.3.2 Internet Protocol (IP)

Quando si parla di *Internet Protocol* (IP) si fa in genere riferimento ad una versione (la quarta, da cui il nome IPv4) che è quella più usata comunemente, anche se ormai si sta diffondendo sempre di più la nuova versione IPv6. Il protocollo IPv4 venne standardizzato nel 1981 dall'RFC 719.

Il protocollo IP (indipendentemente dalla versione) nasce per disaccoppiare le applicazioni della struttura hardware delle reti di trasmissione, e creare una interfaccia di trasmissione dei dati indipendente dal sottostante substrato di interconnessione fisica, che può essere realizzato con le tecnologie più disparate (Ethernet, Token Ring, FDDI, ecc.). Il compito di IP è pertanto quello di trasmettere i pacchetti da un computer all'altro della rete; le caratteristiche essenziali con cui questo viene realizzato in sono due:

- *Universal addressing* la comunicazione avviene fra due stazioni remote identificate univocamente con un indirizzo a 32 bit che può appartenere ad una sola interfaccia di rete.
- *Best effort* viene assicurato il massimo impegno nella trasmissione, ma non c'è nessuna garanzia per i livelli superiori né sulla percentuale di successo né sul tempo di consegna dei pacchetti di dati.

Negli anni '90 la crescita vertiginosa del numero di macchine connesse a Internet ha iniziato a far emergere i vari limiti di IPv4, per risolverne i problemi si è perciò definita una nuova versione del protocollo, che (saltando un numero) è diventata la versione 6. IPv6 nasce quindi come evoluzione di IPv4, mantenendone inalterate le funzioni che si sono dimostrate valide, eliminando quelle inutili e aggiungendone poche altre per mantenere il protocollo il più snello e veloce possibile.

I cambiamenti apportati sono comunque notevoli e si possono essere riassunti a grandi linee nei seguenti punti:

- l'espansione delle capacità di indirizzamento e instradamento, per supportare una gerarchia con più livelli di indirizzamento, un numero di nodi indirizzabili molto maggiore e una auto-configurazione degli indirizzi.
- l'introduzione un nuovo tipo di indirizzamento, l'*anycast* che si aggiunge agli usuali *unicast* e *multicast*.
- la semplificazione del formato dell'intestazione (*header*) dei pacchetti, eliminando o rendendo opzionali alcuni dei campi di IPv4, per eliminare la necessità di rielaborazione della stessa da parte dei router e contenere l'aumento di dimensione dovuto all'ampliamento degli indirizzi.
- un supporto per le opzioni migliorato, per garantire una trasmissione più efficiente del traffico normale, limiti meno stringenti sulle dimensioni delle opzioni, e la flessibilità necessaria per introdurne di nuove in futuro.
- il supporto per delle capacità di *qualità di servizio* (QoS) che permettano di identificare gruppi di dati per i quali si può provvedere un trattamento speciale (in vista dell'uso di Internet per applicazioni multimediali e/o "real-time").

Maggiori dettagli riguardo a caratteristiche, notazioni e funzionamento del protocollo IP sono forniti nell'appendice sez. B.1.

13.3.3 User Datagram Protocol (UDP)

Il protocollo UDP è un protocollo di trasporto molto semplice; la sua descrizione completa è contenuta nell'RFC 768, ma in sostanza esso è una semplice interfaccia al protocollo IP dal livello di trasporto. Quando un'applicazione usa UDP essa scrive un pacchetto di dati (il cosiddetto *datagram* che dà il nome al protocollo) su un socket, al pacchetto viene aggiunto un header molto semplice (per una descrizione più accurata vedi sez. C.2), e poi viene passato al livello superiore (IPv4 o IPv6 che sia) che lo spedisce verso la destinazione. Dato che né IPv4 né IPv6 garantiscono l'affidabilità niente assicura che il pacchetto arrivi a destinazione, né che più pacchetti arrivino nello stesso ordine in cui sono stati spediti.

Pertanto il problema principale che si affronta quando si usa UDP è la mancanza di affidabilità, se si vuole essere sicuri che i pacchetti arrivino a destinazione occorrerà provvedere con l'applicazione, all'interno della quale si dovrà inserire tutto quanto necessario a gestire la notifica di ricevimento, la ritrasmissione, il timeout.

Si tenga conto poi che in UDP niente garantisce che i pacchetti arrivino nello stesso ordine in cui sono stati trasmessi, e può anche accadere che i pacchetti vengano duplicati nella trasmissione, e non solo perduti. Di tutto questo di nuovo deve tenere conto l'applicazione.

Un altro aspetto di UDP è che se un pacchetto raggiunge correttamente la destinazione esso viene passato all'applicazione ricevente in tutta la sua lunghezza, la trasmissione avviene perciò per *record* la cui lunghezza viene anche essa trasmessa all'applicazione all'atto del ricevimento.

Infine UDP è un protocollo che opera senza connessione (*connectionless*) in quanto non è necessario stabilire nessun tipo di relazione tra origine e destinazione dei pacchetti. Si hanno così situazioni in cui un client può scrivere su uno stesso socket pacchetti destinati a server diversi, o un server ricevere su un socket pacchetti provenienti da client diversi. Il modo più semplice di immaginarsi il funzionamento di UDP è quello della radio, in cui si può *trasmettere* e *ricevere* da più stazioni usando la stessa frequenza.

Nonostante gli evidenti svantaggi comportati dall'inaffidabilità UDP ha il grande pregio della velocità, che in certi casi è essenziale; inoltre si presta bene per le applicazioni in cui la connessione non è necessaria, e costituirebbe solo un peso in termini di prestazioni, mentre una perdita di pacchetti può essere tollerata: ad esempio le applicazioni di streaming e quelle che usano il *multicast*.

13.3.4 Transport Control Protocol (TCP)

Il TCP è un protocollo molto complesso, definito nell'RFC 739 e completamente diverso da UDP; alla base della sua progettazione infatti non stanno semplicità e velocità, ma la ricerca della massima affidabilità possibile nella trasmissione dei dati.

La prima differenza con UDP è che TCP provvede sempre una connessione diretta fra un client e un server, attraverso la quale essi possono comunicare; per questo il paragone più appropriato per questo protocollo è quello del collegamento telefonico, in quanto prima viene stabilita una connessione fra due i due capi della comunicazione su cui poi effettuare quest'ultima.

Caratteristica fondamentale di TCP è l'affidabilità; quando i dati vengono inviati attraverso una connessione ne viene richiesto un "ricevuto" (il cosiddetto *acknowledgment*), se questo non arriva essi verranno ritrasmessi per un determinato numero di tentativi, intervallati da un periodo di tempo crescente, fino a che sarà considerata fallita o caduta la connessione (e sarà generato un errore di *timeout*); il periodo di tempo dipende dall'implementazione e può variare far i quattro e i dieci minuti.

Inoltre, per tenere conto delle diverse condizioni in cui può trovarsi la linea di comunicazione, TCP comprende anche un algoritmo di calcolo dinamico del tempo di andata e ritorno dei pacchetti fra un client e un server (il cosiddetto RTT, *Round Trip Time*), che lo rende in grado di adattarsi alle condizioni della rete per non generare inutili ritrasmissioni o cadere facilmente in timeout.

Inoltre TCP è in grado di preservare l'ordine dei dati assegnando un numero di sequenza ad ogni byte che trasmette. Ad esempio se un'applicazione scrive 3000 byte su un socket TCP, questi potranno essere spezzati dal protocollo in due segmenti (le unità di dati passate da TCP a IP vengono chiamate *segment*) di 1500 byte, di cui il primo conterrà il numero di sequenza 1 – 1500 e il secondo il numero 1501 – 3000. In questo modo anche se i segmenti arrivano a destinazione in un ordine diverso, o se alcuni arrivano più volte a causa di ritrasmissioni dovute alla perdita degli *acknowledgment*, all'arrivo sarà comunque possibile riordinare i dati e scartare i duplicati.

Il protocollo provvede anche un controllo di flusso (*flow control*), cioè specifica sempre all'altro capo della trasmissione quanti dati può ricevere tramite una *advertised window* (letteralmente "*finestra annunciata*"), che indica lo spazio disponibile nel buffer di ricezione, cosicché nella trasmissione non vengano inviati più dati di quelli che possono essere ricevuti.

Questa finestra cambia dinamicamente diminuendo con la ricezione dei dati dal socket ed aumentando con la lettura di quest'ultimo da parte dell'applicazione, se diventa nulla il buffer di ricezione è pieno e non verranno accettati altri dati. Si noti che UDP non provvede niente di tutto ciò per cui nulla impedisce che vengano trasmessi pacchetti ad un ritmo che il ricevente non può sostenere.

Infine attraverso TCP la trasmissione è sempre bidirezionale (in inglese si dice che è *full-duplex*). È cioè possibile sia trasmettere che ricevere allo stesso tempo, il che comporta che quanto dicevamo a proposito del controllo di flusso e della gestione della sequenzialità dei dati viene effettuato per entrambe le direzioni di comunicazione.

13.3.5 Limiti e dimensioni riguardanti la trasmissione dei dati

Un aspetto di cui bisogna tenere conto nella programmazione di rete, e che ritornerà in seguito, quando tratteremo gli aspetti più avanzati, è che ci sono una serie di limiti a cui la trasmissione dei dati attraverso i vari livelli del protocollo deve sottostare; limiti che è opportuno tenere presente perché in certi casi si possono avere delle conseguenze sul comportamento delle applicazioni.

Un elenco di questi limiti, insieme ad un breve accenno alle loro origini ed alle eventuali implicazioni che possono avere, è il seguente:

- La dimensione massima di un pacchetto IP è di 65535 byte, compresa l'intestazione. Questo è dovuto al fatto che la dimensione è indicata da un campo apposito nell'header di IP che è lungo 16 bit (vedi fig. B.1).
- La dimensione massima di un pacchetto normale di IPv6 è di 65575 byte; il campo apposito nell'header infatti è sempre a 16 bit, ma la dimensione dell'header è fissa e di 40 byte e non è compresa nel valore indicato dal suddetto campo. Inoltre IPv6 ha la possibilità di estendere la dimensione di un pacchetto usando la *jumbo payload option*.
- Molte reti fisiche hanno una MTU (*Maximum Transfer Unit*) che dipende dal protocollo specifico usato al livello di connessione fisica. Il più comune è quello di ethernet che è pari a 1500 byte, una serie di altri valori possibili sono riportati in tab. 13.3.

Quando un pacchetto IP viene inviato su una interfaccia di rete e le sue dimensioni eccedono la MTU viene eseguita la cosiddetta *frammentazione*, i pacchetti cioè vengono suddivisi in blocchi più piccoli che possono essere trasmessi attraverso l'interfaccia.³

Rete	MTU
Hyperlink	65535
Token Ring IBM (16 Mbit/sec)	17914
Token Ring IEEE 802.5 (4 Mbit/sec)	4464
FDDI	4532
Ethernet	1500
X.25	576

Tabella 13.3: Valori della MTU (*Maximum Transfer Unit*) per una serie di diverse tecnologie di rete.

La MTU più piccola fra due stazioni viene in genere chiamata *path MTU*, che dice qual è la lunghezza massima oltre la quale un pacchetto inviato da una stazione ad un'altra verrebbe senz'altro frammentato. Si tenga conto che non è affatto detto che la *path MTU* sia la stessa in entrambe le direzioni, perché l'instradamento può essere diverso nei due sensi, con diverse tipologie di rete coinvolte.

Una delle differenze fra IPv4 e IPv6 è che per IPv6 la frammentazione può essere eseguita solo alla sorgente, questo vuol dire che i router IPv6 non frammentano i pacchetti che ritrasmettono (anche se possono frammentare i pacchetti che generano loro stessi), al contrario di quanto fanno i router IPv4. In ogni caso una volta frammentati i pacchetti possono essere riassemblati solo alla destinazione.

Nell'header di IPv4 è previsto il flag DF che specifica che il pacchetto non deve essere frammentato; un router che riceva un pacchetto le cui dimensioni eccedano quelle dell'MTU

³questo accade sia per IPv4 che per IPv6, anche se i pacchetti frammentati sono gestiti con modalità diverse, IPv4 usa un flag nell'header, IPv6 una opportuna opzione, si veda sez. B.2.

della rete di destinazione genererà un messaggio di errore ICMPv4 di tipo *destination unreachable, fragmentation needed but DF bit set*. Dato che i router IPv6 non possono effettuare la frammentazione la ricezione di un pacchetto di dimensione eccessiva per la ritrasmissione genererà sempre un messaggio di errore ICMPv6 di tipo *packet too big*.

Dato che il meccanismo di frammentazione e riassemblaggio dei pacchetti comporta inefficienza, normalmente viene utilizzato un procedimento, detto *path MTU discovery* che permette di determinare il *path MTU* fra due stazioni; per la realizzazione del procedimento si usa il flag DF di IPv4 e il comportamento normale di IPv6 inviando delle opportune serie di pacchetti (per i dettagli vedere l’RFC 1191 per IPv4 e l’RFC 1981 per IPv6) fintanto che non si hanno più errori.

Il TCP usa sempre questo meccanismo, che per le implementazioni di IPv4 è opzionale, mentre diventa obbligatorio per IPv6. Per IPv6 infatti, non potendo i router frammentare i pacchetti, è necessario, per poter comunicare, conoscere da subito il *path MTU*.

Infine il TCP definisce una *Maximum Segment Size* o MSS (vedi sez. C.1) che annuncia all’altro capo della connessione la dimensione massima del segmento di dati che può essere ricevuto, così da evitare la frammentazione. Di norma viene impostato alla dimensione della MTU dell’interfaccia meno la lunghezza delle intestazioni di IP e TCP, in Linux il default, mantenuto nella costante TCP_MSS è 512.

Capitolo 14

I socket

In questo capitolo inizieremo a spiegare le caratteristiche salienti della principale interfaccia per la programmazione di rete, quella dei *socket*, che, pur essendo nata in ambiente Unix, è usata ormai da tutti i sistemi operativi.

Dopo una breve panoramica sulle caratteristiche di questa interfaccia vedremo come creare un socket e come collegarlo allo specifico protocollo di rete che si utilizzerà per la comunicazione. Per evitare un'introduzione puramente teorica concluderemo il capitolo con un primo esempio di applicazione.

14.1 Introduzione ai socket

In questa sezione daremo descrizione essenziale di cosa sono i *socket* e di quali sono i concetti fondamentali da tenere presente quando si ha a che fare con essi; ne illustreremo poi le caratteristiche e le differenti tipologie presenti ed infine tratteremo le modalità con cui possono essere creati.

14.1.1 Cosa sono i *socket*

I *socket* (una traduzione letterale potrebbe essere *presa*, ma essendo universalmente noti come *socket* utilizzeremo sempre la parola inglese) sono uno dei principali meccanismi di comunicazione utilizzato in ambito Unix, e li abbiamo brevemente incontrati in sez. 11.1.5, fra i vari meccanismi di intercomunicazione fra processi.

Un socket costituisce in sostanza un canale di comunicazione fra due processi su cui si possono leggere e scrivere dati analogo a quello di una *pipe* (vedi sez. 11.1.1) ma, a differenza di questa e degli altri meccanismi esaminati nel capitolo cap. 11, i socket non sono limitati alla comunicazione fra processi che girano sulla stessa macchina, ma possono realizzare la comunicazione anche attraverso la rete.

Quella dei socket costituisce infatti la principale interfaccia usata nella programmazione di rete. La loro origine risale al 1983, quando furono introdotti in BSD 4.2; l'interfaccia è rimasta sostanzialmente la stessa, con piccole modifiche, negli anni successivi. Benché siano state sviluppate interfacce alternative, originate dai sistemi SVr4 come la XTI (*X/Open Transport Interface*) nessuna ha mai raggiunto la diffusione e la popolarità di quella dei socket (né tantomeno la stessa usabilità e flessibilità) ed oggi sono praticamente dimenticate.

La flessibilità e la genericità dell'interfaccia inoltre consente di utilizzare i socket con i più disparati meccanismi di comunicazione, e non solo con l'insieme dei protocolli TCP/IP, anche se questa sarà comunque quella di cui tratteremo in maniera più estesa.

Per capire il funzionamento dei socket occorre avere presente il funzionamento dei protocolli di rete che su utilizzeranno (ed in particolare quelli del TCP/IP già illustrati in sez. 13.3), ma l'interfaccia è del tutto generale e benché le problematiche, e quindi le modalità di risolvere i problemi, siano diverse a seconda del tipo di protocollo di comunicazione usato, le funzioni da usare nella gestione dei socket restano le stesse.

Per questo motivo una semplice descrizione dell'interfaccia è assolutamente inutile, in quanto il comportamento di quest'ultima e le problematiche da affrontare cambiano radicalmente a seconda del tipo di comunicazione usato. La scelta di questo tipo di comunicazione (sovente anche detto *stile*) va infatti ad incidere sulla semantica che verrà utilizzata a livello utente per gestire la comunicazione cioè su come inviare e ricevere i dati e sul comportamento effettivo delle funzioni utilizzate.

La scelta di uno *stile* dipende sia dai meccanismi disponibili, sia dal tipo di comunicazione che si vuole effettuare. Ad esempio alcuni tipi di comunicazione considerano i dati come una sequenza continua di byte, in quello che viene chiamato un *flusso* (in inglese *stream*), mentre altri invece li raggruppano in *pacchetti* (in inglese *datagram*) che vengono sempre inviati in blocchi separati e non divisibili.

Un altro esempio delle differenze fra i diversi tipi di comunicazione concerne la possibilità che essa possa o meno perdere dati nella trasmissione, che possa o meno rispettare l'ordine in cui i dati inviati e ricevuti, o che possa accadere di inviare dei pacchetti di dati più volte (differenze che ad esempio sono presenti nel caso di utilizzo dei protocolli TCP o UDP).

Un terzo esempio di differenza nel tipo di comunicazione concerne il modo in cui essa avviene nei confronti dei corrispondenti, in certi casi essa può essere condotta con una connessione diretta con un solo corrispondente, come per una telefonata; altri casi possono prevedere una comunicazione come per lettera, in cui si scrive l'indirizzo su ogni pacchetto, altri ancora una comunicazione uno a molti come il *broadcast* ed il *multicast*, in cui i pacchetti possono venire emessi su appositi “canali” dove chiunque si collega possa riceverli.

È chiaro che ciascuno di questi diversi aspetti è associato ad un tipo di comunicazione che comporta una modalità diversa di gestire la stessa, ad esempio se la comunicazione è inaffidabile occorrerà essere in grado di gestire la perdita o il rimescolamento dei dati, se è a pacchetti questi dovranno essere opportunamente trattati, se è uno a molti occorrerà tener conto della eventuale unidirezionalità della stessa, ecc.

14.1.2 La creazione di un socket

Come accennato l'interfaccia dei socket è estremamente flessibile e permette di interagire con protocolli di comunicazione anche molto diversi fra di loro; in questa sezione vedremo come è possibile creare un socket e come specificare il tipo di comunicazione che esso deve utilizzare.

La creazione di un socket avviene attraverso l'uso della funzione di sistema `socket`; essa restituisce un *file descriptor* (del tutto analogo a quelli che si ottengono per i file di dati e le *pipe*, descritti in sez. 5.1.1) che serve come riferimento al socket; il suo prototipo è:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

Apri un socket.

La funzione ritorna un valore positivo in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES non si hanno privilegi per creare un socket nel dominio o con il protocollo specificato.

EAFNOSUPPORT famiglia di indirizzi non supportata.

EINVAL argomento `type` invalido.

EMFILE si è ecceduta la tabella dei file.

ENFILE si è raggiunto il limite massimo di file aperti.

ENOBUFS non c'è sufficiente memoria per creare il socket (può essere anche **ENOMEM**).

EPROTONOSUPPORT il tipo di socket o il protocollo scelto non sono supportati nel dominio.

ed inoltre a seconda del protocollo usato, potranno essere generati altri errori, che sono riportati nelle pagine di manuale relative al protocollo.

La funzione ha tre argomenti, **domain** specifica il dominio del socket (definisce cioè, come vedremo in sez. 14.1.3, la famiglia di protocolli usata), **type** specifica il tipo di socket (definisce cioè, come vedremo in sez. 14.1.4, lo stile di comunicazione) e **protocol** il protocollo; in genere quest'ultimo è indicato implicitamente dal tipo di socket, per cui di norma questo valore viene messo a zero (con l'eccezione dei *raw socket*).

Si noti che la creazione del socket si limita ad allocare le opportune strutture nel kernel (sostanzialmente una voce nella *file table*) e non comporta nulla riguardo all'indicazione degli indirizzi remoti o locali attraverso i quali si vuole effettuare la comunicazione. Questo significa che la funzione da sola non è in grado di fornire alcun tipo di comunicazione.

14.1.3 Il dominio dei socket

Dati i tanti e diversi protocolli di comunicazione disponibili, esistono vari tipi di socket, che vengono classificati raggruppandoli in quelli che si chiamano *domini*. La scelta di un dominio equivale in sostanza alla scelta di una famiglia di protocolli, e viene effettuata attraverso l'argomento **domain** della funzione **socket**. Ciascun dominio ha un suo nome simbolico che convenzionalmente è indicato da una costante che inizia per **PF_**, sigla che sta per *protocol family*, altro nome con cui si indicano i domini.

A ciascun tipo di dominio corrisponde un analogo nome simbolico, anch'esso associato ad una costante, che inizia invece per **AF_** (da *address family*) che identifica il formato degli indirizzi usati in quel dominio. Le pagine di manuale di Linux si riferiscono a questi indirizzi anche come *name space*, (nome che invece il manuale della *glibc* riserva a quello che noi abbiamo chiamato domini) dato che identificano il formato degli indirizzi usati in quel dominio per identificare i capi della comunicazione.

L'idea alla base della distinzione fra questi due insiemi di costanti era che una famiglia di protocolli potesse supportare vari tipi di indirizzi, per cui il prefisso **PF_** si sarebbe dovuto usare nella creazione dei socket e il prefisso **AF_** in quello delle strutture degli indirizzi. Questo è quanto specificato anche dallo standard POSIX.1g, ma non esistono a tuttora famiglie di protocolli che supportino diverse strutture di indirizzi, per cui nella pratica questi due nomi sono equivalenti e corrispondono agli stessi valori numerici.¹ Qui si sono indicati i nomi con il prefisso **AF_** seguendo la convenzione usata nelle pagine di manuale.

I domini (e i relativi nomi simbolici), così come i nomi delle famiglie di indirizzi, sono definiti dall'*header file* **socket.h**. Un elenco, aggiornato alla versione 4.15, delle famiglie di

¹in Linux, come si può verificare andando a guardare il contenuto di `bits/socket.h`, le costanti sono esattamente le stesse e ciascuna **AF_** è definita alla corrispondente **PF_** e con lo stesso nome.

Nome	Valore	Utilizzo	Man page
AF_UNSPEC	0	Non specificato	
AF_LOCAL	1	Local communication	unix(7)
AF_UNIX, AF_FILE	1	Sinonimi di AF_LOCAL	
AF_INET	2	IPv4 Internet protocols	ip(7)
AF_AX25	3	Amateur radio AX.25 protocol	
AF_IPX	4	IPX - Novell protocols	
AF_APPLETALK	5	Appletalk	ddp(7)
AF_NETROM	6	Amateur radio NetROM	
AF_BRIDGE	7	Multiprotocol bridge	
AF_ATMPVC	8	Access to raw ATM PVCs	
AF_X25	9	ITU-T X.25 / ISO-8208 protocol	x25(7)
AF_INET6	10	IPv6 Internet protocols	ipv6(7)
AF_ROSE	11	Amateur Radio X.25 PLP	
AF_DECnet	12	Reserved for DECnet project	
AF_NETBEUI	13	Reserved for 802.2LLC project	
AF_SECURITY	14	Security callback pseudo AF	
AF_KEY	15	AF_KEY key management API	
AF_NETLINK	16	Kernel user interface device	netlink(7)
AF_ROUTE	16	Sinonimo di AF_NETLINK emula BSD.	
AF_PACKET	17	Low level packet interface	packet(7)
AF_ASH	18	Ash	
AF_ECONET	19	Acorn Econet	
AF_ATMSVC	20	ATM SVCs	
AF_RDS	21	RDS Sockets	
AF_SNA	22	Linux SNA Project	
AF_IRDA	23	IRDA socket (infrarossi)	irda(7)
AF_PPPOX	24	PPPoX socket	
AF_WANPIPE	25	Wanpipe API socket	
AF_LLC	26	Linux LLC	
AF_IB	27	Native InfiniBand address	
AF_MPLS	28	MPSL	
AF_CAN	29	Controller Area Network	
AF_TIPC	30	TIPC sockets	
AF_BLUETOOTH	31	Bluetooth socket	
AF_IUCV	32	IUCV sockets	
AF_RXRPC	33	RxRPC sockets	
AF_ISDN	34	mISDN sockets	
AF_PHONET	35	Phonet sockets	
AF_IEEE802154	36	IEEE802154 sockets	
AF_CAIF	37	CAIF sockets	
AF_ALG	38	Algorithm sockets	
AF_NFC	39	NFC sockets	
AF_VSOCK	40	vSockets	
AF_KCM	41	Kernel Connection Multiplexor	
AF_QIPCRTR	42	Qualcomm IPC Router	
AF_SMC	43	smc sockets	

Tabella 14.1: Famiglie di protocolli definiti in Linux.

protocolli disponibili in Linux è riportato in tab. 14.1. L'elenco indica tutti i protocolli definiti; fra questi però saranno utilizzabili solo quelli per i quali si è compilato il supporto nel kernel (o si sono caricati gli opportuni moduli), viene definita anche una costante `AF_MAX` che indica il valore massimo associabile ad un dominio.

Si tenga presente che non tutte le famiglie di protocolli sono utilizzabili dall'utente generico, ad esempio in generale tutti i socket di tipo `SOCK_RAW` possono essere creati solo da processi che hanno i privilegi di amministratore (cioè con `UID` effettivo uguale a zero) o dotati della *capability* `CAP_NET_RAW`.

14.1.4 Il tipo di socket

La scelta di un dominio non comporta però la scelta dello stile di comunicazione, questo infatti viene a dipendere dal protocollo che si andrà ad utilizzare fra quelli disponibili nella famiglia scelta. L'interfaccia dei socket permette di scegliere lo stile di comunicazione indicando il tipo di socket con l'argomento `type` di `socket`. Linux mette a disposizione vari tipi di socket (che corrispondono a quelli che il manuale della *glibc* [?] chiama *styles*) identificati dalle seguenti costanti:²

- SOCK_STREAM** Provvede un canale di trasmissione dati bidirezionale, sequenziale e affidabile. Opera su una connessione con un altro socket. I dati vengono ricevuti e trasmessi come un flusso continuo di byte (da cui il nome *stream*) e possono essere letti in blocchi di dimensioni qualunque. Può supportare la trasmissione dei cosiddetti dati urgenti (o *out-of-band*, vedi sez. 18.1.4).
- SOCK_DGRAM** Viene usato per trasmettere pacchetti di dati (*datagram*) di lunghezza massima prefissata, indirizzati singolarmente. Non esiste una connessione e la trasmissione è effettuata in maniera non affidabile.
- SOCK_SEQPACKET** Provvede un canale di trasmissione di dati bidirezionale, sequenziale e affidabile. Opera su una connessione con un altro socket. I dati possono vengono trasmessi per pacchetti di dimensione massima fissata, e devono essere letti integralmente da ciascuna chiamata a `read`.
- SOCK_RAW** Provvede l'accesso a basso livello ai protocolli di rete e alle varie interfacce. I normali programmi di comunicazione non devono usarlo, è riservato all'uso di sistema.
- SOCK_RDM** Provvede un canale di trasmissione di dati affidabile, ma in cui non è garantito l'ordine di arrivo dei pacchetti.
- SOCK_PACKET** Obsoleto, non deve essere più usato (e pertanto non ne parleremo ulteriormente).

A partire dal kernel 2.6.27 l'argomento `type` della funzione `socket` assume un significato ulteriore perché può essere utilizzato per impostare dei flag relativi alle caratteristiche generali del *socket* non strettamente attinenti all'indicazione del tipo secondo i valori appena illustrati. Essi infatti possono essere combinati con un OR aritmetico delle ulteriori costanti:

- SOCK_CLOEXEC** imposta il flag di *close-on-exec* sul file descriptor del socket, ottenendo lo stesso effetto del flag `O_CLOEXEC` di `open` (vedi tab. 5.4), di cui costituisce l'analogo.
- SOCK_NONBLOCK** crea il socket in modalità non-bloccante, con effetti identici ad una successiva chiamata a `fcntl` per impostare il flag di `O_NONBLOCK` sul file descriptor (si faccia di nuovo riferimento al significato di quest'ultimo come spiegato in tab. 5.4).

Si tenga presente inoltre che non tutte le combinazioni fra una famiglia di protocolli e un tipo di socket sono valide, in quanto non è detto che in una famiglia esista un protocollo per ciascuno dei diversi stili di comunicazione appena elencati.

²le pagine di manuale POSIX riportano solo i primi tre tipi, Linux supporta anche gli altri, come si può verificare nel file `include/linux/net.h` dei sorgenti del kernel.

Famiglia	Tipo				
	SOCK_STREAM	SOCK_DGRAM	SOCK_RAW	SOCK_RDM	SOCK_SEQPACKET
AF_UNIX	sì	sì	—	—	sì ³
AF_LOCAL	sinonimo di AF_UNIX				
AF_INET	TCP	UDP	IPv4	—	—
AF_INET6	TCP	UDP	IPv6	—	—
AF_IPX	—	sì	—	—	—
AF_NETLINK	—	sì	sì	—	—
AF_X25	—	—	—	—	sì
AF_AX25	—	sì	sì	—	sì
AF_APPLETALK	—	sì	sì	—	—
AF_PACKET	—	sì	sì	—	—
AF_KEY	—	—	sì	—	—
AF_IRDA	sì	sì	sì	—	sì
AF_NETROM	—	—	—	—	sì
AF_ROSE	—	—	—	—	sì
AF_RDS	—	—	—	—	sì
AF_ECONET	—	sì	—	—	—

Tabella 14.2: Combinazioni valide di dominio e tipo di protocollo per la funzione `socket`.

In tab. 14.2 sono mostrate le combinazioni valide possibili per le principali famiglie di protocolli. Per ogni combinazione valida si è indicato il tipo di protocollo, o la parola *si* qualora il protocollo non abbia un nome definito, mentre si sono lasciate vuote le caselle per le combinazioni non supportate.

14.2 Le strutture degli indirizzi dei socket

Come si è visto nella creazione di un socket non si specifica nulla oltre al tipo di famiglia di protocolli che si vuole utilizzare, in particolare nessun indirizzo che identifichi i due capi della comunicazione. La funzione infatti si limita ad allocare nel kernel quanto necessario per poter poi realizzare la comunicazione.

Gli indirizzi infatti vengono specificati attraverso apposite strutture che vengono utilizzate dalle altre funzioni della interfaccia dei socket, quando la comunicazione viene effettivamente realizzata. Ogni famiglia di protocolli ha ovviamente una sua forma di indirizzamento e in corrispondenza a questa una sua peculiare struttura degli indirizzi. I nomi di tutte queste strutture iniziano per `sockaddr_`; quelli propri di ciascuna famiglia vengono identificati dal suffisso finale, aggiunto al nome precedente.

14.2.1 La struttura generica

Le strutture degli indirizzi vengono sempre passate alle varie funzioni attraverso puntatori (cioè *by reference*), ma le funzioni devono poter maneggiare puntatori a strutture relative a tutti gli indirizzi possibili nelle varie famiglie di protocolli; questo pone il problema di come passare questi puntatori, il C moderno risolve questo problema coi i puntatori generici (i `void *`), ma l'interfaccia dei socket è antecedente alla definizione dello standard ANSI C, e per questo nel 1982 fu scelto di definire una struttura generica per gli indirizzi dei socket, `sockaddr`, che si è riportata in fig. 14.1.

³supportati a partire dal kernel 2.6.4 per socket che conservano i limiti dei messaggi e li consegnano in sequenza ordinata.

```

struct sockaddr {
    sa_family_t  sa_family;    /* address family: AF_XXX */
    char         sa_data[14];  /* address (protocol-specific) */
};

```

Figura 14.1: La struttura generica degli indirizzi dei socket `sockaddr`.

Tutte le funzioni dei socket che usano gli indirizzi sono definite usando nel prototipo un puntatore a questa struttura; per questo motivo quando si invocano dette funzioni passando l'indirizzo di un protocollo specifico occorrerà eseguire una conversione del relativo puntatore.

I tipi di dati che compongono la struttura sono stabiliti dallo standard POSIX.1g e li abbiamo riassunti in tab. 14.3 con i rispettivi file di include in cui sono definiti; la struttura è invece definita nell'include file `sys/socket.h`.

Tipo	Descrizione	Header
<code>int8_t</code>	intero a 8 bit con segno	<code>sys/types.h</code>
<code>uint8_t</code>	intero a 8 bit senza segno	<code>sys/types.h</code>
<code>int16_t</code>	intero a 16 bit con segno	<code>sys/types.h</code>
<code>uint16_t</code>	intero a 16 bit senza segno	<code>sys/types.h</code>
<code>int32_t</code>	intero a 32 bit con segno	<code>sys/types.h</code>
<code>uint32_t</code>	intero a 32 bit senza segno	<code>sys/types.h</code>
<code>sa_family_t</code>	famiglia degli indirizzi	<code>sys/socket.h</code>
<code>socklen_t</code>	lunghezza (<code>uint32_t</code>) dell'indirizzo di un socket	<code>sys/socket.h</code>
<code>in_addr_t</code>	indirizzo IPv4 (<code>uint32_t</code>)	<code>netinet/in.h</code>
<code>in_port_t</code>	porta TCP o UDP (<code>uint16_t</code>)	<code>netinet/in.h</code>

Tabella 14.3: Tipi di dati usati nelle strutture degli indirizzi, secondo quanto stabilito dallo standard POSIX.1g.

In alcuni sistemi la struttura è leggermente diversa e prevede un primo membro aggiuntivo `uint8_t sin_len` (come riportato da R. Stevens in [?]). Questo campo non verrebbe usato direttamente dal programmatore e non è richiesto dallo standard POSIX.1g, in Linux pertanto non esiste. Il campo `sa_family_t` era storicamente un `unsigned short`.

Dal punto di vista del programmatore l'unico uso di questa struttura è quello di fare da riferimento per il casting, per il kernel le cose sono un po' diverse, in quanto esso usa il puntatore per recuperare il campo `sa_family`, comune a tutte le famiglie, con cui determinare il tipo di indirizzo; per questo motivo, anche se l'uso di un puntatore `void *` sarebbe più immediato per l'utente (che non dovrebbe più eseguire il casting), è stato mantenuto l'uso di questa struttura.

Se si usa una struttura `sockaddr` per allocare delle variabili generiche da usare in seguito per degli indirizzi si pone il problema che niente assicura che i dati necessari per le varie famiglie di indirizzi possano rientrare nella dimensione del campo `sa_data` indicata in fig. 14.1, anzi, come vedremo in sez. 14.2.3, nel caso di indirizzi IPv6 questa non è proprio sufficiente.

Per questo l'interfaccia di programmazione dei socket prevede la definizione di una speciale struttura `sockaddr_storage` illustrata in fig. 14.2, in cui di nuovo si usa il primo campo (`ss_family`) per indicare il tipo di indirizzo, ed in cui i campi successivi sono utilizzati per allineare i dati al tipo di architettura hardware utilizzata, e per allocare uno spazio sufficiente ampio per contenere qualunque tipo di indirizzo supportato. Allocando questa struttura si ha la certezza di non eccedere le dimensioni qualunque sia il tipo di indirizzi che si useranno,

```

struct sockaddr_storage {
    sa_family_t  ss_family;           /* address family: AF_XXX */
    __ss_align_t __ss_align;
    char         __ss_padding[_SS_PADSIZE]; /* address (protocol-specific) */
};

```

Figura 14.2: La struttura generica degli indirizzi dei socket `sockaddr_storage`.

pertanto risulta utile tutte le volte che si devono gestire in maniera generica tipi di indirizzi diversi (ad esempio IPv4 ed IPv6).

14.2.2 La struttura degli indirizzi IPv4

I socket di tipo `AF_INET` vengono usati per la comunicazione attraverso Internet; la struttura per gli indirizzi per un socket Internet (se si usa IPv4) è definita come `sockaddr_in` nell'header file `netinet/in.h` ed ha la forma mostrata in fig. 14.3, conforme allo standard POSIX.1g.

```

struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr  sin_addr;   /* internet address */
};

/* Internet address. */
struct in_addr {
    in_addr_t      s_addr;     /* address in network byte order */
};

```

Figura 14.3: La struttura `sockaddr_in` degli indirizzi dei socket Internet (IPv4) e la struttura `in_addr` degli indirizzi IPv4.

L'indirizzo di un socket Internet (secondo IPv4) comprende l'indirizzo Internet di un'interfaccia più un *numero di porta* (affronteremo in dettaglio il significato di questi numeri in sez. 15.1.6). Il protocollo IP di per sé non prevede numeri di porta, questi sono utilizzati solo dai protocolli di livello superiore come TCP e UDP, ma devono essere indicati qui. Inoltre questa struttura viene usata anche per i socket RAW che accedono direttamente al livello di IP, in questo caso il numero della porta deve essere impostato al numero di protocollo.

Il membro `sin_family` deve essere sempre impostato a `AF_INET`, altrimenti si avrà un errore di `EINVAL`; il membro `sin_port` specifica il *numero di porta*. I numeri di porta sotto il 1024 sono chiamati *riservati* in quanto utilizzati da servizi standard e soltanto processi con i privilegi di amministratore (con `UID` effettivo uguale a zero) o con la *capability* `CAP_NET_BIND_SERVICE` possono usare la funzione `bind` (che vedremo in sez. 15.2.1) su queste porte.

Il membro `sin_addr` contiene un indirizzo Internet, e viene acceduto sia come struttura (un resto di una implementazione precedente in cui questa era una *union* usata per accedere alle diverse classi di indirizzi) che direttamente come intero. In `netinet/in.h` vengono definite anche alcune costanti che identificano alcuni indirizzi speciali, riportati in tab. 15.1, che rincontreremo più avanti.

Infine occorre sottolineare che sia gli indirizzi che i numeri di porta devono essere specificati in quello che viene chiamato *network order*, cioè con i bit ordinati in formato *big endian*

(vedi sez. 2.4.4), questo comporta la necessità di usare apposite funzioni di conversione per mantenere la portabilità del codice (vedi sez. 14.3 per i dettagli del problema e le relative soluzioni).

14.2.3 La struttura degli indirizzi IPv6

Essendo IPv6 un'estensione di IPv4, i socket di tipo `AF_INET6` sono sostanzialmente identici ai precedenti; la parte in cui si trovano praticamente tutte le differenze fra i due socket è quella della struttura degli indirizzi; la sua definizione, presa da `netinet/in.h`, è riportata in fig. 14.4.

```

struct sockaddr_in6 {
    sa_family_t    sin6_family;    /* AF_INET6 */
    in_port_t      sin6_port;      /* port number */
    uint32_t       sin6_flowinfo;  /* IPv6 flow information */
    struct in6_addr sin6_addr;      /* IPv6 address */
    uint32_t       sin6_scope_id;  /* Scope id (new in 2.4) */
};
struct in6_addr {
    uint8_t        s6_addr[16];    /* IPv6 address */
};

```

Figura 14.4: La struttura `sockaddr_in6` degli indirizzi dei socket IPv6 e la struttura `in6_addr` degli indirizzi IPv6.

Il campo `sin6_family` deve essere sempre impostato ad `AF_INET6`, il campo `sin6_port` è analogo a quello di IPv4 e segue le stesse regole; il campo `sin6_flowinfo` è a sua volta diviso in tre parti di cui i 24 bit inferiori indicano l'etichetta di flusso, i successivi 4 bit la priorità e gli ultimi 4 sono riservati. Questi valori fanno riferimento ad alcuni campi specifici dell'header dei pacchetti IPv6 (vedi sez. B.2.3) ed il loro uso è sperimentale.

Il campo `sin6_addr` contiene l'indirizzo a 128 bit usato da IPv6, espresso da un vettore di 16 byte; anche in questo caso esistono alcuni valori predeterminati, ma essendo il campo un vettore di byte non è possibile assegnarli con il calore di una costante. Esistono però le variabili predefinite `in6addr_any` (che indica l'indirizzo generico) e `in6addr_loopback` (che indica l'indirizzo di loopback) il cui valore può essere copiato in questo campo. A queste due variabili si aggiungono le macro `IN6ADDR_ANY_INIT` e `IN6ADDR_LOOPBACK_INIT` per effettuare delle assegnazioni statiche.

Infine il campo `sin6_scope_id` è un campo introdotto in Linux con il kernel 2.4, per gestire alcune operazioni riguardanti il *multicasting*, è supportato solo per gli indirizzi di tipo *link-local* (vedi sez. B.2.7) e deve contenere l'*interface index* (vedi sez. 16.3.2) della scheda di rete. Si noti infine che `sockaddr_in6` ha una dimensione maggiore della struttura `sockaddr` generica di fig. 14.1, quindi occorre stare attenti a non avere fatto assunzioni riguardo alla possibilità di contenere i dati nelle dimensioni di quest'ultima (per questo se necessario è opportuno usare `sockaddr_storage`).

14.2.4 La struttura degli indirizzi locali

I socket di tipo `AF_UNIX` o `AF_LOCAL` vengono usati per una comunicazione fra processi che stanno sulla stessa macchina (per questo vengono chiamati *local domain* o anche *Unix domain*); essi hanno la caratteristica ulteriore di poter essere creati anche in maniera anonima

attraverso la funzione `socketpair` (che abbiamo trattato in sez. 11.1.5). Quando però si vuole fare riferimento esplicito ad uno di questi socket si deve usare una struttura degli indirizzi di tipo `sockaddr_un`, la cui definizione si è riportata in fig. 14.5.

```
#define UNIX_PATH_MAX    108
struct sockaddr_un {
    sa_family_t    sun_family;          /* AF_UNIX */
    char            sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

Figura 14.5: La struttura `sockaddr_un` degli indirizzi dei socket locali (detti anche *unix domain*) definita in `sys/un.h`.

In questo caso il campo `sun_family` deve essere `AF_UNIX`, mentre il campo `sun_path` deve specificare un indirizzo. Questo ha due forme; può essere “*named*” ed in tal caso deve corrispondere ad un file (di tipo socket) presente nel filesystem o essere “*abstract*” nel qual caso viene identificato da una stringa univoca in uno spazio di nomi astratto.

Nel primo caso l’indirizzo viene specificato in `sun_path` come una stringa (terminata da uno zero) corrispondente al *pathname* del file; nel secondo caso (che è specifico di Linux e non portabile) `sun_path` deve iniziare con uno zero ed il nome verrà costituito dai restanti byte che verranno interpretati come stringa senza terminazione (un byte nullo non ha in questo caso nessun significato).

In realtà esiste una terza forma, *unnamed*, che non è possibile indicare in fase di scrittura, ma che è quella che viene usata quando si legge l’indirizzo di un socket anonimo creato con `socketpair`; in tal caso la struttura restituita è di dimensione `sizeof(sa_family_t)`, quindi `sun_path` non esiste e non deve essere referenziato.

14.2.5 La struttura degli indirizzi AppleTalk

I socket di tipo `AF_APPLETALK` sono usati dalla libreria `netatalk` per implementare la comunicazione secondo il protocollo AppleTalk, uno dei primi protocolli di rete usato nel mondo dei personal computer, usato dalla Apple per connettere fra loro computer e stampanti. Il kernel supporta solo due strati del protocollo, DDP e AARP, e di norma è opportuno usare le funzioni della libreria `netatalk`, tratteremo qui questo argomento principalmente per mostrare l’uso di un protocollo alternativo.

I socket AppleTalk permettono di usare il protocollo DDP, che è un protocollo a pacchetto, di tipo `SOCK_DGRAM`; l’argomento `protocol` di `socket` deve essere nullo. È altresì possibile usare i socket raw specificando un tipo `SOCK_RAW`, nel qual caso l’unico valore valido per `protocol` è `ATPROTO_DDP`.

Gli indirizzi AppleTalk devono essere specificati tramite una struttura `sockaddr_atalk`, la cui definizione è riportata in fig. 14.6; la struttura viene dichiarata includendo il file `netatalk/at.h`.

Il campo `sat_family` deve essere sempre `AF_APPLETALK`, mentre il campo `sat_port` specifica la porta che identifica i vari servizi. Valori inferiori a 129 sono usati per le *porte riservate*, e possono essere usati solo da processi con i privilegi di amministratore o con la *capability* `CAP_NET_BIND_SERVICE`.

L’indirizzo remoto è specificato nella struttura `sat_addr`, e deve essere in *network order* (vedi sez. 2.4.4); esso è composto da una parte di rete data dal campo `s_net`, che può assumere il valore `AT_ANYNET`, che indica una rete generica e vale anche per indicare la rete su cui si è,

```

struct sockaddr_atalk {
    sa_family_t    sat_family; /* address family */
    uint8_t        sat_port;   /* port */
    struct at_addr  sat_addr;   /* net/node */
};
struct at_addr {
    uint16_t       s_net;
    uint8_t        s_node;
};

```

Figura 14.6: La struttura `sockaddr_atalk` degli indirizzi dei socket AppleTalk, e la struttura `at_addr` degli indirizzi AppleTalk.

il singolo nodo è indicato da `s_node`, e può prendere il valore generico `AT_ANYNODE` che indica anche il nodo corrente, ed il valore `ATADDR_BCAST` che indica tutti i nodi della rete.

14.2.6 La struttura degli indirizzi dei *packet socket*

I *packet socket*, identificati dal dominio `AF_PACKET`, sono un'interfaccia specifica di Linux per inviare e ricevere pacchetti direttamente su un'interfaccia di rete, senza passare per le funzioni di gestione dei protocolli di livello superiore. In questo modo è possibile implementare dei protocolli in *user space*, agendo direttamente sul livello fisico. In genere comunque si preferisce usare la libreria `pcap`,⁴ che assicura la portabilità su altre piattaforme, anche se con funzionalità ridotte.

Questi socket possono essere di tipo `SOCK_RAW` o `SOCK_DGRAM`. Con socket di tipo `SOCK_RAW` si può operare sul livello di collegamento, ed i pacchetti vengono passati direttamente dal socket al driver del dispositivo e viceversa. In questo modo, in fase di trasmissione, il contenuto completo dei pacchetti, comprese le varie intestazioni, deve essere fornito dall'utente. In fase di ricezione invece tutto il contenuto del pacchetto viene passato inalterato sul socket, anche se il kernel analizza comunque il pacchetto, riempiendo gli opportuni campi della struttura `sockaddr_ll` ad esso associata.

Si usano invece socket di tipo `SOCK_DGRAM` quando si vuole operare a livello di rete.

In questo caso in fase di ricezione l'intestazione del protocollo di collegamento viene rimossa prima di passare il resto del pacchetto all'utente, mentre in fase di trasmissione viene creata una opportuna intestazione per il protocollo a livello di collegamento utilizzato, usando le informazioni necessarie che devono essere specificate sempre con una struttura `sockaddr_ll`.

Nella creazione di un *packet socket* il valore dell'argomento `protocol` di `socket` serve a specificare, in *network order*, il numero identificativo del protocollo di collegamento si vuole utilizzare. I valori possibili sono definiti secondo lo standard IEEE 802.3, e quelli disponibili in Linux sono accessibili attraverso opportune costanti simboliche definite nel file `linux/if_ether.h`. Se si usa il valore speciale `ETH_P_ALL` passeranno sul *packet socket* tutti i pacchetti, qualunque sia il loro protocollo di collegamento. Ovviamente l'uso di questi socket è una operazione privilegiata e può essere effettuati solo da un processo con i privilegi di amministratore (*UID* effettivo nullo) o con la *capability* `CAP_NET_RAW`.

Una volta aperto un *packet socket*, tutti i pacchetti del protocollo specificato passeranno attraverso di esso, qualunque sia l'interfaccia da cui provengono; se si vuole limitare il passag-

⁴la libreria è mantenuta insieme al comando `tcpdump`, informazioni e documentazione si possono trovare sul sito del progetto <http://www.tcpdump.org/>.

gio ad una interfaccia specifica occorre usare la funzione `bind` (vedi sez. 15.2.1) per agganciare il socket a quest'ultima.

```

struct sockaddr_ll {
    unsigned short  sll_family;    /* Always AF_PACKET */
    unsigned short  sll_protocol; /* Physical layer protocol */
    int             sll_ifindex;   /* Interface number */
    unsigned short  sll_hatype;   /* Header type */
    unsigned char   sll_pkttype;  /* Packet type */
    unsigned char   sll_halen;    /* Length of address */
    unsigned char   sll_addr[8];  /* Physical layer address */
};

```

Figura 14.7: La struttura `sockaddr_ll` degli indirizzi dei *packet socket*.

Nel caso dei *packet socket* la struttura degli indirizzi è di tipo `sockaddr_ll`, e la sua definizione è riportata in fig. 14.7; essa però viene ad assumere un ruolo leggermente diverso rispetto a quanto visto finora per gli altri tipi di socket. Infatti se il socket è di tipo `SOCK_RAW` si deve comunque scrivere tutto direttamente nel pacchetto, quindi la struttura non serve più a specificare gli indirizzi. Essa mantiene questo ruolo solo per i socket di tipo `SOCK_DGRAM`, per i quali permette di specificare i dati necessari al protocollo di collegamento, mentre viene sempre utilizzata in lettura (per entrambi i tipi di socket), per la ricezione dei dati relativi a ciascun pacchetto.

Al solito il campo `sll_family` deve essere sempre impostato al valore `AF_PACKET`. Il campo `sll_protocol` indica il protocollo scelto, e deve essere indicato in *network order*, facendo uso delle costanti simboliche definite in `linux/if_ether.h`. Il campo `sll_ifindex` è l'indice dell'interfaccia (l'*interface index* (vedi sez. 16.3.2) che in caso di presenza di più interfacce dello stesso tipo (se ad esempio si hanno più schede Ethernet), permette di selezionare quella con cui si vuole operare (un valore nullo indica qualunque interfaccia). Questi sono i due soli campi che devono essere specificati quando si vuole selezionare una interfaccia specifica, usando questa struttura con la funzione `bind`.

I campi `sll_halen` e `sll_addr` indicano rispettivamente l'indirizzo associato all'interfaccia sul protocollo di collegamento e la relativa lunghezza; ovviamente questi valori cambiano a seconda del tipo di collegamento che si usa, ad esempio, nel caso di Ethernet, questi saranno il MAC address della scheda e la relativa lunghezza. Essi vengono usati, insieme ai campi `sll_family` e `sll_ifindex` quando si inviano dei pacchetti, in questo caso tutti gli altri campi devono essere nulli.

Il campo `sll_hatype` indica il tipo ARP, come definito in `linux/if_arp.h`, mentre il campo `sll_pkttype` indica il tipo di pacchetto; entrambi vengono impostati alla ricezione di un pacchetto ed han senso solo in questo caso. In particolare `sll_pkttype` può assumere i seguenti valori: `PACKET_HOST` per un pacchetto indirizzato alla macchina ricevente, `PACKET_BROADCAST` per un pacchetto di *broadcast*, `PACKET_MULTICAST` per un pacchetto inviato ad un indirizzo fisico di *multicast*, `PACKET_OTHERHOST` per un pacchetto inviato ad un'altra stazione (e ricevuto su un'interfaccia in modo promiscuo), `PACKET_OUTGOING` per un pacchetto originato dalla propria macchina che torna indietro sul socket.

Si tenga presente infine che in fase di ricezione, anche se si richiede il troncamento del pacchetto, le funzioni `recv`, `recvfrom` e `recvmsg` (vedi sez. 18.1.2) restituiranno comunque la lunghezza effettiva del pacchetto così come arrivato sulla linea.

14.3 Le funzioni di conversione degli indirizzi

In questa sezione tratteremo delle varie funzioni usate per manipolare gli indirizzi, limitandoci però agli indirizzi Internet. Come accennato gli indirizzi e i numeri di porta usati nella rete devono essere forniti nel cosiddetto *network order*, che corrisponde al formato *big endian* (vedi sez. 2.4.4), anche quando la proprio macchina non usa questo formato, cosa che può comportare la necessità di eseguire delle conversioni.

14.3.1 Le funzioni per il riordinamento

Come già visto in sez. 2.4.4 il problema connesso all'*endianness* è che quando si passano dei dati da un tipo di architettura all'altra i dati vengono interpretati in maniera diversa, e ad esempio nel caso dell'intero a 16 bit ci si ritroverà con i due byte in cui è suddiviso scambiati di posto.

Per questo motivo si usano delle funzioni di conversione che servono a tener conto automaticamente della possibile differenza fra l'ordinamento usato sul computer e quello che viene usato nelle trasmissioni sulla rete; queste funzioni sono `htonl`, `htons`, `ntohl` e `ntohs` ed i rispettivi prototipi sono:

```
#include <arpa/inet.h>
unsigned long int htonl(unsigned long int hostlong)
    Converte l'intero a 32 bit hostlong dal formato della macchina a quello della rete.
unsigned short int htons(unsigned short int hostshort)
    Converte l'intero a 16 bit hostshort dal formato della macchina a quello della rete.
unsigned long int ntohl(unsigned long int netlong)
    Converte l'intero a 32 bit netlong dal formato della rete a quello della macchina.
unsigned short int ntohs(unsigned short int netshort)
    Converte l'intero a 16 bit netshort dal formato della rete a quello della macchina.
```

Tutte le funzioni restituiscono il valore convertito, e non prevedono errori.

I nomi sono assegnati usando la lettera *n* come mnemonico per indicare l'ordinamento usato sulla rete (da *network order*) e la lettera *h* come mnemonico per l'ordinamento usato sulla macchina locale (da *host order*), mentre le lettere *s* e *l* stanno ad indicare i tipi di dato (*long* o *short*, riportati anche dai prototipi).

Usando queste funzioni si ha la conversione automatica: nel caso in cui la macchina che si sta usando abbia una architettura *big endian* queste funzioni sono definite come macro che non fanno nulla. Per questo motivo vanno sempre utilizzate, anche quando potrebbero non essere necessarie, in modo da assicurare la portabilità del codice su tutte le architetture.

14.3.2 Le funzioni di conversione per gli indirizzi IPv4

Un secondo insieme di funzioni di manipolazione è quello che serve per passare dalla rappresentazione simbolica degli indirizzi IP al formato binario previsto dalla struttura degli indirizzi di fig. 14.3, e viceversa. La notazione più comune è la cosiddetta notazione *dotted-decimal*, che prevede che gli indirizzi IPv4 siano indicati con l'espressione del valore numerico decimale di ciascuno dei 4 byte che li costituiscono separati da un punto (ad esempio 192.168.0.1).

In realtà le funzioni che illustreremo supportano una notazione che più propriamente dovrebbe esser chiamata *numbers-and-dot* in quanto il valore può essere indicato con numeri espressi sia in decimale, che in ottale (se indicati apponendo uno zero) che in esadecimale (se indicati apponendo 0x). Inoltre per la parte meno significativa dell'espressione, quella che

riguarda l'indirizzo locale, si può usare, eliminando altrettanti punti, valori a 16 o a 24 bit, e togliendo tutti i punti, si può usare anche direttamente un valore numerico a 32 bit.⁵

Tradizionalmente la conversione di un indirizzo *dotted-decimal* al valore numerico veniva eseguita dalla funzione `inet_addr` (prevista fin dalle origini in BSD e inclusa in POSIX.1-2001) il cui prototipo è:

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *strptr)
    Converta la stringa dell'indirizzo dotted decimal in nel numero IP in network order.
```

La funzione ritorna il valore dell'indirizzo in caso di successo e `INADDR_NONE` per un errore e non genera codici di errore.

La prima funzione, `inet_addr`, restituisce l'indirizzo a 32 bit in *network order* (del tipo `in_addr_t`) a partire dalla stringa passata nell'argomento `strptr`. In caso di errore (quando la stringa non esprime un indirizzo valido) restituisce invece il valore `INADDR_NONE`, che tipicamente sono trentadue bit a uno. Questo però comporta che la stringa `255.255.255.255`, che pure è un indirizzo valido, non può essere usata con questa funzione dato che genererebbe comunque un errore; per questo motivo essa è generalmente deprecata in favore di `inet_aton`.

Per effettuare la conversione inversa la funzione usata tradizionalmente è `inet_ntoa`, anch'essa presente fin da BSD 4.3, in cui si riprende la notazione già vista in sez. 14.3.1 che usa la lettera `n` come mnemonico per indicare la rete ed `a` (per ASCII) come mnemonico per indicare la stringa corrispondente all'indirizzo; il suo prototipo è:

```
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr addrptr)
    Converta un indirizzo IP in una stringa dotted decimal.
```

La funzione l'indirizzo della stringa con il valore dell'indirizzo convertito e non prevede errori.

La funzione converte il valore a 32 bit dell'indirizzo, espresso in *network order*, e preso direttamente con un puntatore al relativo campo della struttura degli indirizzi, restituendo il puntatore alla stringa che contiene l'espressione in formato *dotted-decimal*. Si deve tenere presente che la stringa risiede in un segmento di memoria statica, per cui viene riscritta ad ogni chiamata e la funzione non è rientrante.

Per rimediare ai problemi di `inet_addr` è stata sostituita da `inet_aton`, che però non è stata standardizzata e non è presente in POSIX.1-2001, anche se è definita sulla gran parte dei sistemi Unix; il suo prototipo è:

```
#include <arpa/inet.h>
int inet_aton(const char *src, struct in_addr *dest)
    Converta la stringa dell'indirizzo dotted decimal in un indirizzo IP.
```

La funzione ritorna un valore non nullo in caso di successo e 0 per un errore e non genera codici di errore.

La funzione converte la stringa puntata da `src` nell'indirizzo binario che viene memorizzato nell'opportuna struttura `in_addr` (si veda fig. 14.3) situata all'indirizzo dato dall'argomento `dest` (è espressa in questa forma in modo da poterla usare direttamente con il puntatore usato per passare la struttura degli indirizzi). La funzione restituisce un valore diverso da zero se l'indirizzo è valido e la conversione ha successo e 0 in caso contrario. Se usata con `dest` inizializzato a `NULL` può essere usata per effettuare la validazione dell'indirizzo espresso da `src`.

⁵la funzionalità si trova anche in gran parte dei programmi che usano indirizzi di rete, e deriva direttamente da queste funzioni.

Oltre a queste tre funzioni esistono le ulteriori `inet_lnaof`, `inet_netof` e `inet_makeaddr` che assumono la ormai obsoleta e deprecata suddivisione in classi degli indirizzi IP per fornire la parte di rete e quella di indirizzo locale. Ad oggi il loro uso non ha più alcun senso per ciò non le tratteremo.

14.3.3 Le funzioni di conversione per indirizzi IP generici

Le tre funzioni precedenti sono limitate solo ad indirizzi IPv4, per questo motivo è preferibile usare le due nuove funzioni `inet_pton` e `inet_ntop` che possono convertire anche gli indirizzi IPv6. Anche in questo caso le lettere `n` e `p` sono degli mnemonici per ricordare il tipo di conversione effettuata e stanno per *presentation* e *numeric*.

Entrambe le funzioni accettano l'argomento `af` che indica il tipo di indirizzo, e che può essere soltanto `AF_INET` o `AF_INET6`. La prima funzione, `inet_pton`, serve a convertire una stringa in un indirizzo; il suo prototipo è:

```
#include <sys/socket.h>
int inet_pton(int af, const char *src, void *addr_ptr)
    Converta l'indirizzo espresso tramite una stringa nel valore numerico.
```

La funzione ritorna 1 in caso di successo, 0 se `src` non contiene una rappresentazione valida per la famiglia di indirizzi indicati da `af` e `-1` se `af` specifica una famiglia di indirizzi non valida, e solo in quest'ultimo caso `errno` assumerà il valore `EAFNOSUPPORT`.

La funzione converte la stringa indicata tramite `src` nel valore numerico dell'indirizzo IP del tipo specificato da `af` che viene memorizzato all'indirizzo puntato da `addr_ptr`. La funzione supporta per IPv4 la sola notazione *dotted-decimal*, e non quella più completa *number-and-dot* che abbiamo visto per `inet_aton`. Per IPv6 la notazione prevede la suddivisione dei 128 bit dell'indirizzo in 16 parti di 16 bit espresse con valori esadecimali separati dal carattere ":" ed una serie di valori nulli possono essere sostituiti (una sola volta, sempre a partire dalla sinistra) con la notazione "::", un esempio di indirizzo in questa forma potrebbe essere `2001:db8::8:ba98:2078:e3e3`, per una descrizione più completa si veda sez. B.2.5.

La seconda funzione di conversione è `inet_ntop` che converte un indirizzo in una stringa; il suo prototipo è:

```
#include <sys/socket.h>
char *inet_ntop(int af, const void *addr_ptr, char *dest, size_t len)
    Converta l'indirizzo dalla relativa struttura in una stringa simbolica.
```

La funzione ritorna un puntatore non nullo alla stringa convertita in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà uno dei valori:

`ENOSPC` le dimensioni della stringa con la conversione dell'indirizzo eccedono la lunghezza specificata da `len`.

`ENOAFSUPPORT` la famiglia di indirizzi `af` non è una valida.

La funzione converte la struttura dell'indirizzo puntata da `addr_ptr` in una stringa che viene copiata nel buffer puntato dall'indirizzo `dest`; questo deve essere preallocato dall'utente e la lunghezza deve essere almeno `INET_ADDRSTRLEN` in caso di indirizzi IPv4 e `INET6_ADDRSTRLEN` per indirizzi IPv6; la lunghezza del buffer deve comunque essere specificata con il parametro `len`.

Gli indirizzi vengono convertiti da/alle rispettive strutture di indirizzo (una struttura `in_addr` per IPv4, e una struttura `in6_addr` per IPv6), che devono essere precedentemente allocate e passate attraverso il puntatore `addr_ptr`; l'argomento `dest` di `inet_ntop` non può essere nullo e deve essere allocato precedentemente.

Capitolo 15

I socket TCP

In questo capitolo tratteremo le basi dei socket TCP, iniziando con una descrizione delle principali caratteristiche del funzionamento di una connessione TCP; vedremo poi le varie funzioni che servono alla creazione di una connessione fra client e server, fornendo alcuni esempi elementari, e finiremo prendendo in esame l'uso dell'*I/O multiplexing*.

15.1 Il funzionamento di una connessione TCP

Prima di entrare nei dettagli delle singole funzioni usate nelle applicazioni che utilizzano i socket TCP, è fondamentale spiegare alcune delle basi del funzionamento del protocollo, poiché questa conoscenza è essenziale per comprendere il comportamento di dette funzioni per questo tipo di socket, ed il relativo modello di programmazione.

Si ricordi che il protocollo TCP serve a creare degli *stream socket*, cioè una forma di canale di comunicazione che stabilisce una connessione stabile fra due macchine in rete, in modo che queste possano scambiarsi dei dati. In questa sezione ci concentreremo sulle modalità con le quali il protocollo dà inizio e conclude una connessione e faremo inoltre un breve accenno al significato di alcuni dei vari *stati* ad essa associati.

15.1.1 La creazione della connessione: il *three way handshake*

Il processo che porta a creare una connessione TCP viene chiamato *three way handshake*; la successione tipica degli eventi (e dei *segmenti*¹ di dati che vengono scambiati) che porta alla creazione di una connessione è la seguente:

1. Il server deve essere preparato per accettare le connessioni in arrivo; il procedimento si chiama *apertura passiva* del socket (in inglese *passive open*). Questo viene fatto chiamando la sequenza di funzioni `socket`, `bind` e `listen`. Completata l'apertura passiva il server chiama la funzione `accept` e il processo si blocca in attesa di connessioni.
2. Il client richiede l'inizio della connessione usando la funzione `connect`, attraverso un procedimento che viene chiamato *apertura attiva*, dall'inglese *active open*. La chiamata

¹si ricordi che il *segmento* è l'unità elementare di dati trasmessa dal protocollo TCP al livello successivo; tutti i segmenti hanno un'intestazione che contiene le informazioni che servono allo *stack TCP* (così viene di solito chiamata la parte del kernel che realizza il protocollo) per effettuare la comunicazione, fra questi dati ci sono una serie di flag usati per gestire la connessione, come SYN, ACK, URG, FIN, alcuni di essi, come SYN (che sta per *synchronize*) corrispondono a funzioni particolari del protocollo e danno il nome al segmento, (per maggiori dettagli si veda sez. C.1).

di `connect` blocca il processo e causa l'invio da parte del client di un segmento SYN, in sostanza viene inviato al server un pacchetto IP che contiene solo le intestazioni di IP e TCP (con il numero di sequenza iniziale e il flag SYN) e le opzioni di TCP.

3. Il server deve dare ricevuto (l'*acknowledge*) del SYN del client, inoltre anche il server deve inviare il suo SYN al client (e trasmettere il suo numero di sequenza iniziale) questo viene fatto ritrasmettendo un singolo segmento in cui sono impostati entrambi i flag SYN e ACK.
4. Una volta che il client ha ricevuto l'*acknowledge* dal server la funzione `connect` ritorna, l'ultimo passo è dare il ricevuto del SYN del server inviando un ACK. Alla ricezione di quest'ultimo la funzione `accept` del server ritorna e la connessione è stabilita.

Il procedimento viene chiamato *three way handshake* dato che per realizzarlo devono essere scambiati tre segmenti. In fig. 15.1 si è rappresentata graficamente la sequenza di scambio dei segmenti che stabilisce la connessione.

Figura 15.1: Il *three way handshake* del TCP.

Si è accennato in precedenza ai *numeri di sequenza*, che sono anche riportati in fig. 15.1: per gestire una connessione affidabile infatti il protocollo TCP prevede nell'intestazione la presenza di un numero a 32 bit (chiamato appunto *sequence number*) che identifica a quale byte nella sequenza del flusso corrisponde il primo byte della sezione dati contenuta nel segmento.

Il numero di sequenza di ciascun segmento viene calcolato a partire da un *numero di sequenza iniziale* generato in maniera casuale del kernel all'inizio della connessione e trasmesso con il SYN; l'*acknowledgement* di ciascun segmento viene effettuato dall'altro capo della connessione impostando il flag ACK e restituendo nell'apposito campo dell'intestazione un *acknowledge number* pari al numero di sequenza che il ricevente si aspetta di ricevere con il pacchetto successivo; dato che il primo pacchetto SYN consuma un byte, nel *three way handshake* il numero di *acknowledge* è sempre pari al numero di sequenza iniziale incrementato di uno; lo stesso varrà anche (vedi fig. 15.2) per l'*acknowledgement* di un segmento FIN.

15.1.2 Le opzioni TCP.

Ciascun segmento SYN contiene in genere delle opzioni per il protocollo TCP, le cosiddette *TCP options*, da non confondere con le opzioni dei socket TCP che tratteremo in sez. 16.2.5. In questo caso infatti si tratta delle opzioni che vengono trasmesse come parte di un pacchetto TCP, e non delle funzioni che consentono di impostare i relativi valori. Queste opzioni vengono inserite fra l'intestazione ed i dati, e servono a comunicare all'altro capo una serie di parametri utili a regolare la connessione. Normalmente vengono usate le seguenti opzioni:

- *MSS option*, con questa opzione ciascun capo della connessione annuncia all'altro il massimo ammontare di dati (MMS sta appunto per *Maximum Segment Size*, vedi sez. C.1) che vorrebbe accettare per ciascun segmento nella connessione corrente. È possibile leggere e scrivere questo valore attraverso l'opzione del socket `TCP_MAXSEG` (vedi sez. 16.2.5).
- *window scale option*, il protocollo TCP realizza il controllo di flusso attraverso una *advertised window* (la “finestra annunciata”, vedi sez. C.1) con la quale ciascun capo della comunicazione dichiara quanto spazio disponibile ha in memoria per i dati. Questo è un numero a 16 bit dell'header, che così può indicare un massimo di 65535 byte² ma

²in Linux il massimo è 32767 per evitare problemi con alcune realizzazioni dello *stack TCP* che usano l'aritmetica con segno.

alcuni tipi di connessione come quelle ad alta velocità (sopra i 45Mbit/sec) e quelle che hanno grandi ritardi nel cammino dei pacchetti (come i satelliti) richiedono una finestra più grande per poter ottenere il massimo dalla trasmissione.

Per questo esiste un'altra opzione che indica un fattore di scala da applicare al valore della finestra annunciata per la connessione corrente, espresso come numero di bit cui spostare a sinistra il valore della finestra annunciata inserito nel pacchetto. Essendo una nuova opzione per garantire la compatibilità con delle vecchie realizzazioni del protocollo la procedura che la attiva prevede come negoziazione che l'altro capo della connessione riconosca esplicitamente l'opzione inserendola anche lui nel suo SYN di risposta dell'apertura della connessione.

Con Linux è possibile indicare al kernel di far negoziare il fattore di scala in fase di creazione di una connessione tramite la `sysctl tcp_window_scaling` (vedi sez. 16.4.3). Per poter usare questa funzionalità è comunque necessario ampliare le dimensioni dei buffer di ricezione e spedizione, cosa che può essere fatta sia a livello di sistema con le opportune `sysctl` (vedi sez. 16.4.3) che a livello di singoli socket con le relative opzioni (vedi sez. 16.2.5).

- *timestamp option*, è anche questa una nuova opzione necessaria per le connessioni ad alta velocità per evitare possibili corruzioni di dati dovute a pacchetti perduti che riappaiono; anche questa viene negoziata all'inizio della connessione come la precedente.

La *MSS option* è generalmente supportata da quasi tutte le realizzazioni del protocollo, le ultime due opzioni (trattate nell'RFC 1323) sono meno comuni; vengono anche dette *long fat pipe options* dato che questo è il nome che viene dato alle connessioni caratterizzate da alta velocità o da ritardi elevati. In ogni caso Linux supporta pienamente entrambe queste opzioni aggiuntive.

15.1.3 La terminazione della connessione

Mentre per la creazione di una connessione occorre un interscambio di tre segmenti, la procedura di chiusura ne richiede normalmente quattro. In questo caso la successione degli eventi è la seguente:

1. Un processo ad uno dei due capi chiama la funzione `close`, dando l'avvio a quella che viene chiamata *chiusura attiva* (o *active close*). Questo comporta l'emissione di un segmento FIN, che serve ad indicare che si è finito con l'invio dei dati sulla connessione.
2. L'altro capo della connessione riceve il segmento FIN e dovrà eseguire la *chiusura passiva* (o *passive close*). Al FIN, come ad ogni altro pacchetto, viene risposto con un ACK, inoltre il ricevimento del FIN viene segnalato al processo che ha aperto il socket (dopo che ogni altro eventuale dato rimasto in coda è stato ricevuto) come un *end-of-file* sulla lettura: questo perché il ricevimento di un segmento FIN significa che non si riceveranno altri dati sulla connessione.
3. Una volta rilevata l'*end-of-file* anche il secondo processo chiamerà la funzione `close` sul proprio socket, causando l'emissione di un altro segmento FIN.
4. L'altro capo della connessione riceverà il segmento FIN conclusivo e risponderà con un ACK.

Dato che in questo caso sono richiesti un FIN ed un ACK per ciascuna direzione normalmente i segmenti scambiati sono quattro. Questo non è vero sempre giacché in alcune situazioni il FIN del passo 1 è inviato insieme a dei dati. Inoltre è possibile che i segmenti inviati nei passi 2 e 3 dal capo che effettua la chiusura passiva, siano accorpati in un singolo

segmento. Come per il SYN anche il FIN occupa un byte nel numero di sequenza, per cui l'ACK riporterà un *acknowledge number* incrementato di uno. In fig. 15.2 si è rappresentata graficamente la sequenza di scambio dei segmenti che conclude la connessione.

Figura 15.2: La chiusura di una connessione TCP.

Si noti che, nella sequenza di chiusura, fra i passi 2 e 3, è in teoria possibile che si mantenga un flusso di dati dal capo della connessione che deve ancora eseguire la chiusura passiva a quello che sta eseguendo la chiusura attiva. Nella sequenza indicata i dati verrebbero persi, dato che si è chiuso il socket dal lato che esegue la chiusura attiva; esistono tuttavia situazioni in cui si vuole poter sfruttare questa possibilità, usando una procedura che è chiamata *half-close*; torneremo su questo aspetto e su come utilizzarlo in sez. 15.6.3, quando parleremo della funzione *shutdown*.

La emissione del segmento FIN avviene quando il socket viene chiuso, questo però non avviene solo per la chiamata esplicita della funzione *close*, ma anche alla terminazione di un processo, quando tutti i file vengono chiusi. Questo comporta ad esempio che se un processo viene terminato da un segnale tutte le connessioni aperte verranno chiuse.

Infine occorre sottolineare che, benché nella figura (e nell'esempio che vedremo più avanti in sez. 15.4.1) sia stato il client ad eseguire la chiusura attiva, nella realtà questa può essere eseguita da uno qualunque dei due capi della comunicazione (come nell'esempio di fig. 15.9), e che anche se il caso più comune resta quello del client, ci sono alcuni servizi, il più noto dei quali è l'HTTP 1.0 (con le versioni successive il default è cambiato) per i quali è il server ad effettuare la chiusura attiva.

15.1.4 Un esempio di connessione

Come abbiamo visto le operazioni del TCP nella creazione e conclusione di una connessione sono piuttosto complesse, ed abbiamo esaminato soltanto quelle relative ad un andamento normale. In sez. C.1.1 vedremo con maggiori dettagli che una connessione può assumere vari stati, che ne caratterizzano il funzionamento, e che sono quelli che vengono riportati dal comando *netstat*, per ciascun socket TCP aperto, nel campo *State*.

Non possiamo affrontare qui una descrizione completa del funzionamento del protocollo; un approfondimento sugli aspetti principali si trova in sez. C.1, ma per una trattazione completa il miglior riferimento resta [?]. Qui ci limiteremo a descrivere brevemente un semplice esempio di connessione e le transizioni che avvengono nei due casi appena citati (creazione e terminazione della connessione).

In assenza di connessione lo stato del TCP è *CLOSED*; quando una applicazione esegue una apertura attiva il TCP emette un SYN e lo stato diventa *SYN_SENT*; quando il TCP riceve la risposta del SYN+ACK emette un ACK e passa allo stato *ESTABLISHED*; questo è lo stato finale in cui avviene la gran parte del trasferimento dei dati.

Dal lato server in genere invece il passaggio che si opera con l'apertura passiva è quello di portare il socket dallo stato *CLOSED* allo stato *LISTEN* in cui vengono accettate le connessioni.

Dallo stato *ESTABLISHED* si può uscire in due modi; se un'applicazione chiama la funzione *close* prima di aver ricevuto un *end-of-file* (chiusura attiva) la transizione è verso lo stato *FIN_WAIT_1*; se invece l'applicazione riceve un FIN nello stato *ESTABLISHED* (chiusura passiva) la transizione è verso lo stato *CLOSE_WAIT*.

In fig. 15.3 è riportato lo schema dello scambio dei pacchetti che avviene per una un esempio di connessione, insieme ai vari stati che il protocollo viene ad assumere per i due lati, server e client.

Figura 15.3: Schema dello scambio di pacchetti per un esempio di connessione.

La connessione viene iniziata dal client che annuncia una *Maximum Segment Size* di 1460, un valore tipico con Linux per IPv4 su Ethernet, il server risponde con lo stesso valore (ma potrebbe essere anche un valore diverso).

Una volta che la connessione è stabilita il client scrive al server una richiesta (che assumiamo stare in un singolo segmento, cioè essere minore dei 1460 byte annunciati dal server), quest'ultimo riceve la richiesta e restituisce una risposta (che di nuovo supponiamo stare in un singolo segmento). Si noti che l'*acknowledge* della richiesta è mandato insieme alla risposta: questo viene chiamato *piggybacking* ed avviene tutte le volte che il server è sufficientemente veloce a costruire la risposta; in caso contrario si avrebbe prima l'emissione di un ACK e poi l'invio della risposta.

Infine si ha lo scambio dei quattro segmenti che terminano la connessione secondo quanto visto in sez. 15.1.3; si noti che il capo della connessione che esegue la chiusura attiva entra nello stato `TIME_WAIT`, sul cui significato torneremo fra poco.

È da notare come per effettuare uno scambio di due pacchetti (uno di richiesta e uno di risposta) TCP necessiti di ulteriori otto segmenti, se invece si fosse usato UDP sarebbero stati sufficienti due soli pacchetti. Questo è il costo che occorre pagare per avere l'affidabilità garantita da TCP, se si fosse usato UDP si sarebbe dovuto trasferire la gestione di tutta una serie di dettagli (come la verifica della ricezione dei pacchetti) dal livello del trasporto all'interno dell'applicazione.

Quello che è bene sempre tenere presente è allora quali sono le esigenze che si hanno in una applicazione di rete, perché non è detto che TCP sia la miglior scelta in tutti i casi (ad esempio se si devono solo scambiare dati già organizzati in piccoli pacchetti l'overhead aggiunto può essere eccessivo) per questo esistono applicazioni che usano UDP e lo fanno perché nel caso specifico le sue caratteristiche di velocità e compattezza nello scambio dei dati rispondono meglio alle esigenze che devono essere affrontate.

15.1.5 Lo stato `TIME_WAIT`

Come riportato da Stevens in [?] lo stato `TIME_WAIT` è probabilmente uno degli aspetti meno compresi del protocollo TCP, è infatti comune trovare domande su come sia possibile evitare che un'applicazione resti in questo stato lasciando attiva una connessione ormai conclusa; la risposta è che non deve essere fatto, ed il motivo cercheremo di spiegarlo adesso.

Come si è visto nell'esempio precedente (vedi fig. 15.3) `TIME_WAIT` è lo stato finale in cui il capo di una connessione che esegue la chiusura attiva resta prima di passare alla chiusura definitiva della connessione. Il tempo in cui l'applicazione resta in questo stato deve essere due volte la *Maximum Segment Lifetime* (da qui in avanti abbreviata in *MSL*).

La *Maximum Segment Lifetime* è la stima del massimo periodo di tempo in secondi che un pacchetto IP può vivere sulla rete. Questo tempo è limitato perché ogni pacchetto IP può essere ritrasmesso dai router un numero massimo di volte (detto *hop limit*). Il numero di ritrasmissioni consentito è indicato dal campo `TTL` dell'intestazione di IP (per maggiori dettagli vedi sez. B.1), e viene decrementato ad ogni passaggio da un router; quando si annulla il pacchetto viene scartato. Siccome il numero è ad 8 bit il numero massimo di "salti" è di 255, pertanto anche se il `TTL` (da *time to live*) non è propriamente un limite sul tempo, sulla sua base si può stimare che un pacchetto IP non possa restare nella rete per più di un certo numero di secondi, che costituisce la *MSL*.

Ogni realizzazione del TCP deve scegliere un valore per la MSL; l’RFC 1122 raccomanda 2 minuti, mentre Linux usa 30 secondi, questo comporta una durata dello stato `TIME_WAIT` che a seconda delle realizzazioni può variare fra 1 a 4 minuti. Lo stato `TIME_WAIT` viene utilizzato dal protocollo per due motivi principali:

1. effettuare in maniera affidabile la terminazione della connessione in entrambe le direzioni.
2. consentire l’eliminazione dei segmenti duplicati dalla rete.

Il punto è che entrambe le ragioni sono importanti, anche se spesso si fa riferimento solo alla prima; ma è solo se si tiene conto della seconda che si capisce il perché della scelta di un tempo pari al doppio della MSL come durata di questo stato.

Il primo dei due motivi precedenti si può capire tornando a fig. 15.3: assumendo che l’ultimo ACK della sequenza (quello del capo che ha eseguito la chiusura attiva) venga perso, chi esegue la chiusura passiva non ricevendo risposta rimanderà un ulteriore FIN, per questo motivo chi esegue la chiusura attiva deve mantenere lo stato della connessione per essere in grado di reinviare l’ACK e chiuderla correttamente. Se non fosse così la risposta sarebbe un RST (un altro tipo di segmento) che verrebbe interpretato come un errore.

Se il TCP deve poter chiudere in maniera pulita entrambe le direzioni della connessione allora deve essere in grado di affrontare la perdita di uno qualunque dei quattro segmenti che costituiscono la chiusura. Per questo motivo un socket deve rimanere attivo nello stato `TIME_WAIT` anche dopo l’invio dell’ultimo ACK, per potere essere in grado di gestirne l’eventuale ritrasmissione, in caso esso venga perduto.

Il secondo motivo è più complesso da capire, e necessita di una spiegazione degli scenari in cui può accadere che i pacchetti TCP si possano perdere nella rete o restare intrappolati, per poi riemergere in un secondo tempo.

Il caso più comune in cui questo avviene è quello di anomalie nell’instradamento; può accadere cioè che un router smetta di funzionare o che una connessione fra due router si interrompa. In questo caso i protocolli di instradamento dei pacchetti possono impiegare diverso tempo (anche dell’ordine dei minuti) prima di trovare e stabilire un percorso alternativo per i pacchetti. Nel frattempo possono accadere casi in cui un router manda i pacchetti verso un altro e quest’ultimo li rispedisce indietro, o li manda ad un terzo router che li rispedisce al primo, si creano cioè dei circoli (i cosiddetti *routing loop*) in cui restano intrappolati i pacchetti.

Se uno di questi pacchetti intrappolati è un segmento TCP, chi l’ha inviato, non ricevendo un ACK in risposta, provvederà alla ritrasmissione e se nel frattempo sarà stata stabilita una strada alternativa il pacchetto ritrasmesso giungerà a destinazione.

Ma se dopo un po’ di tempo (che non supera il limite dell’MSL, dato che altrimenti verrebbe ecceduto il TTL) l’anomalia viene a cessare, il circolo di instradamento viene spezzato i pacchetti intrappolati potranno essere inviati alla destinazione finale, con la conseguenza di avere dei pacchetti duplicati; questo è un caso che il TCP deve essere in grado di gestire.

Allora per capire la seconda ragione per l’esistenza dello stato `TIME_WAIT` si consideri il caso seguente: si supponga di avere una connessione fra l’IP 195.110.112.236 porta 1550 e l’IP 192.84.145.100 porta 22 (affronteremo il significato delle porte nella prossima sezione), che questa venga chiusa e che poco dopo si ristabilisca la stessa connessione fra gli stessi IP sulle stesse porte (quella che viene detta, essendo gli stessi porte e numeri IP, una nuova *incarnazione* della connessione precedente); in questo caso ci si potrebbe trovare con dei pacchetti duplicati relativi alla precedente connessione che riappaiono nella nuova.

Ma fintanto che il socket non è chiuso una nuova incarnazione non può essere creata: per questo un socket TCP resta sempre nello stato `TIME_WAIT` per un periodo di 2 volte la MSL,

prima per attendere MSL secondi per essere sicuri che tutti i pacchetti duplicati in arrivo siano stati ricevuti e scartati, o che nel frattempo siano stati eliminati dalla rete, poi altri MSL secondi per essere sicuri che lo stesso avvenga per le risposte nella direzione opposta.

In questo modo, prima che venga creata una nuova connessione, il protocollo TCP si assicura che tutti gli eventuali segmenti residui di una precedente connessione, che potrebbero causare disturbi, siano stati eliminati dalla rete.

15.1.6 I numeri di porta

In un ambiente multitasking in un dato momento più processi devono poter usare sia UDP che TCP, e ci devono poter essere più connessioni in contemporanea. Per poter tenere distinte le diverse connessioni entrambi i protocolli usano i *numeri di porta*, che fanno parte, come si può vedere in sez. 14.2.2 e sez. 14.2.3 pure delle strutture degli indirizzi del socket.

Quando un client contatta una macchina server deve poter identificare con quale dei vari possibili programmi server attivi intende parlare. Sia TCP che UDP definiscono un gruppo di *porte conosciute* (le cosiddette *well-known port*) che identificano una serie di servizi noti (ad esempio la porta 22 identifica il servizio SSH) effettuati da appositi programmi server che rispondono alle connessioni verso tali porte.

D'altra parte un client non ha necessità di usare dalla sua parte un numero di porta specifico, per cui in genere vengono usate le cosiddette *porte effimere* (o *ephemeral ports*) cioè porte a cui non è assegnato nessun servizio noto e che vengono assegnate automaticamente dal kernel alla creazione della connessione. Queste sono dette effimere in quanto vengono usate solo per la durata della connessione, e l'unico requisito che deve essere soddisfatto è che ognuna di esse sia assegnata in maniera univoca.

La lista delle porte conosciute è definita dall'RFC 1700 che contiene l'elenco delle porte assegnate dalla IANA (la *Internet Assigned Number Authority*) ma l'elenco viene costantemente aggiornato e pubblicato su Internet (una versione aggiornata si può trovare all'indirizzo <http://www.iana.org/assignments/port-numbers>); inoltre in un sistema unix-like un analogo elenco viene mantenuto nel file `/etc/services`, con la corrispondenza fra i vari numeri di porta ed il nome simbolico del servizio. I numeri sono divisi in tre intervalli:

1. *le porte note*. I numeri da 0 a 1023. Queste sono controllate e assegnate dalla IANA. Se è possibile la stessa porta è assegnata allo stesso servizio sia su UDP che su TCP (ad esempio la porta 22 è assegnata a SSH su entrambi i protocolli, anche se viene usata solo dal TCP).
2. *le porte registrate*. I numeri da 1024 a 49151. Queste porte non sono controllate dalla IANA, che però registra ed elenca chi usa queste porte come servizio agli utenti. Come per le precedenti si assegna una porta ad un servizio sia per TCP che UDP anche se poi il servizio è effettuato solo su TCP. Ad esempio *X Window* usa le porte TCP e UDP dal 6000 al 6063 anche se il protocollo viene usato solo con TCP.
3. *le porte private o dinamiche*. I numeri da 49152 a 65535. La IANA non dice nulla riguardo a queste porte che pertanto sono i candidati naturali ad essere usate come porte effimere.

In realtà rispetto a quanto indicato nell'RFC 1700 i vari sistemi hanno fatto scelte diverse per le porte effimere, in particolare in fig. 15.4 sono riportate quelle di BSD e Linux.

Figura 15.4: Allocazione dei numeri di porta.

I sistemi Unix hanno inoltre il concetto di *porte riservate*, che corrispondono alle porte con numero minore di 1024 e coincidono quindi con le *porte note*. La loro caratteristica è che possono essere assegnate a un socket solo da un processo con i privilegi di amministratore, per far sì che solo l'amministratore possa allocare queste porte per far partire i relativi servizi.

Le *glibc* definiscono in `netinet/in.h` le costanti `IPPORT_RESERVED` e `IPPORT_USERRESERVED`, in cui la prima (che vale 1024) indica il limite superiore delle porte riservate, e la seconda (che vale 5000) il limite inferiore delle porte a disposizione degli utenti. La convenzione vorrebbe che le porte *effimere* siano allocate fra questi due valori. Nel caso di Linux questo è vero solo in uno dei due casi di fig. 15.4, e la scelta fra i due possibili intervalli viene fatta dinamicamente dal kernel a seconda della memoria disponibile per la gestione delle relative tabelle.

Si tenga conto poi che ci sono alcuni client, in particolare `rsh` e `rlogin`, che richiedono una connessione su una porta riservata anche dal lato client come parte dell'autenticazione, contando appunto sul fatto che solo l'amministratore può usare queste porte. Data l'assoluta inconsistenza in termini di sicurezza di un tale metodo, al giorno d'oggi esso è in completo disuso.

Data una connessione TCP, ma la cosa vale anche per altri protocolli del livello di trasporto come UDP, si suole chiamare *socket pair*³ la combinazione dei quattro numeri che definiscono i due capi della connessione e cioè l'indirizzo IP locale e la porta TCP locale, e l'indirizzo IP remoto e la porta TCP remota. Questa combinazione, che scriveremo usando una notazione del tipo (195.110.112.152:22, 192.84.146.100:20100), identifica univocamente una connessione su Internet. Questo concetto viene di solito esteso anche a UDP, benché in questo caso non abbia senso parlare di connessione. L'utilizzo del programma `netstat` permette di visualizzare queste informazioni nei campi *Local Address* e *Foreign Address*.

Per capire meglio l'uso delle porte e come vengono utilizzate quando si ha a che fare con un'applicazione client/server (come quelle che descriveremo in sez. 15.3 e sez. 15.4) esamineremo cosa accade con le connessioni nel caso di un server TCP che deve gestire connessioni multiple.

Se eseguiamo un `netstat` su una macchina di prova (il cui indirizzo sia 195.110.112.152) potremo avere un risultato del tipo:

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp      0      0 0.0.0.0:25              0.0.0.0:*               LISTEN
tcp      0      0 127.0.0.1:53             0.0.0.0:*               LISTEN
```

essendo presenti e attivi un server SSH, un server di posta e un DNS per il caching locale.

Questo ci mostra ad esempio che il server SSH ha compiuto un'apertura passiva, mettendosi in ascolto sulla porta 22 riservata a questo servizio, e che si è posto in ascolto per connessioni provenienti da uno qualunque degli indirizzi associati alle interfacce locali. La notazione `0.0.0.0` usata da `netstat` è equivalente all'asterisco utilizzato per il numero di porta, indica il valore generico, e corrisponde al valore `INADDR_ANY` definito in `arpa/inet.h` (vedi 15.1).

Inoltre si noti come la porta e l'indirizzo di ogni eventuale connessione esterna non sono specificati; in questo caso la *socket pair* associata al socket potrebbe essere indicata come `(*:22, *:*)`, usando anche per gli indirizzi l'asterisco come carattere che indica il valore generico.

Dato che in genere una macchina è associata ad un solo indirizzo IP, ci si può chiedere che senso abbia l'utilizzo dell'indirizzo generico per specificare l'indirizzo locale; ma a parte il caso di macchine che hanno più di un indirizzo IP (il cosiddetto *multihoming*) esiste sempre

³da non confondere con la coppia di socket della omonima funzione `socketpair` di sez. 11.1.5 che fanno riferimento ad una coppia di socket sulla stessa macchina, non ai capi di una connessione TCP.

anche l'indirizzo di *loopback*, per cui con l'uso dell'indirizzo generico si possono accettare connessioni indirizzate verso uno qualunque degli indirizzi IP presenti. Ma, come si può vedere nell'esempio con il DNS che è in ascolto sulla porta 53, è possibile anche restringere l'accesso ad uno specifico indirizzo, cosa che nel caso è fatta accettando solo connessioni che arrivino sull'interfaccia di *loopback*.

Una volta che ci si vorrà collegare a questa macchina da un'altra, per esempio quella con l'indirizzo 192.84.146.100, si dovrà lanciare su quest'ultima un client *ssh* per creare una connessione, e il kernel gli assocerà una porta effimera (ad esempio la 21100), per cui la connessione sarà espressa dalla *socket pair* (192.84.146.100:21100, 195.110.112.152:22).

Alla ricezione della richiesta dal client il server creerà un processo figlio per gestire la connessione, se a questo punto eseguiamo nuovamente il programma *netstat* otteniamo come risultato:

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 0.0.0.0:22             0.0.0.0:*              LISTEN
tcp      0      0 0.0.0.0:25             0.0.0.0:*              LISTEN
tcp      0      0 127.0.0.1:53           0.0.0.0:*              LISTEN
tcp      0      0 195.110.112.152:22     192.84.146.100:21100   ESTABLISHED
```

Come si può notare il server è ancora in ascolto sulla porta 22, però adesso c'è un nuovo socket (con lo stato ESTABLISHED) che utilizza anch'esso la porta 22, ed ha specificato l'indirizzo locale, questo è il socket con cui il processo figlio gestisce la connessione mentre il padre resta in ascolto sul socket originale.

Se a questo punto lanciamo un'altra volta il client *ssh* per una seconda connessione quello che otterremo usando *netstat* sarà qualcosa del genere:

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp      0      0 0.0.0.0:22             0.0.0.0:*              LISTEN
tcp      0      0 0.0.0.0:25             0.0.0.0:*              LISTEN
tcp      0      0 127.0.0.1:53           0.0.0.0:*              LISTEN
tcp      0      0 195.110.112.152:22     192.84.146.100:21100   ESTABLISHED
tcp      0      0 195.110.112.152:22     192.84.146.100:21101   ESTABLISHED
```

cioè il client effettuerà la connessione usando un'altra porta effimera: con questa sarà aperta la connessione, ed il server creerà un altro processo figlio per gestirla.

Tutto ciò mostra come il TCP, per poter gestire le connessioni con un server concorrente, non può suddividere i pacchetti solo sulla base della porta di destinazione, ma deve usare tutta l'informazione contenuta nella *socket pair*, compresa la porta dell'indirizzo remoto. E se andassimo a vedere quali sono i processi (ad esempio con il comando *fuser*, o con *lsof*, o usando l'opzione *-p*) a cui fanno riferimento i vari socket vedremmo che i pacchetti che arrivano dalla porta remota 21100 vanno al primo figlio e quelli che arrivano alla porta 21101 al secondo.

15.2 Le funzioni di base per la gestione dei socket

In questa sezione descriveremo in maggior dettaglio le varie funzioni che vengono usate per la gestione di base dei socket TCP, non torneremo però sulla funzione *socket*, che è già stata esaminata accuratamente nel capitolo precedente in sez. 14.1.2. Pur trattandole principalmente dal punto di vista dei socket TCP, daremo brevemente conto del loro comportamento anche per altri tipi di socket.

15.2.1 La funzione bind

La funzione `bind` assegna un indirizzo locale ad un socket,⁴ ed è usata cioè per specificare la prima parte dalla *socket pair*. Viene usata sul lato server per specificare la porta (e gli eventuali indirizzi locali) su cui poi ci si porrà in ascolto. Il prototipo della funzione è il seguente:

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)
                                     Assegna un indirizzo ad un socket.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES si è cercato di usare una porta riservata senza sufficienti privilegi.

EADDRINUSE qualche altro socket sta già usando l'indirizzo richiesto oppure quando non si è richiesta una porta specifica per avere una porta effimera non ve ne sono di disponibili nell'intervallo ad esse riservato correntemente in uso.

EBADF il file descriptor non è valido.

EINVAL il socket ha già un indirizzo assegnato, o `addrlen` è sbagliata, o `serv_addr` non è valido per il dominio.

ENOTSOCK il file descriptor non è associato ad un socket.

ed **EFAULT** nel suo significato generico; inoltre per i socket di tipo `AF_UNIX`:

EADDRNOTAVAIL il tipo di indirizzo specificato non è disponibile.

ed **ELOOP**, **ENAMETOOLONG**, **ENOENT**, **ENOMEM**, **ENOTDIR** e **EROFS** nel loro significato generico.

Il primo argomento è un file descriptor ottenuto da una precedente chiamata a `socket`, mentre il secondo e terzo argomento sono rispettivamente l'indirizzo (locale) del socket e la dimensione della struttura che lo contiene, secondo quanto già trattato in sez. 14.2.

Con i socket TCP la chiamata `bind` permette di specificare l'indirizzo, la porta, entrambi o nessuno dei due. In genere i server utilizzano una porta nota che assegnano all'avvio, se questo non viene fatto è il kernel a scegliere una porta effimera quando vengono eseguite le funzioni `connect` o `listen`, ma se questo è normale per il client non lo è per il server⁵ che in genere viene identificato dalla porta su cui risponde (l'elenco di queste porte, e dei relativi servizi, è in `/etc/services`).

Con `bind` si può assegnare un indirizzo IP specifico ad un socket, purché questo appartenga ad una interfaccia della macchina. Per un client TCP questo diventerà l'indirizzo sorgente usato per i tutti i pacchetti inviati sul socket, mentre per un server TCP questo restringerà l'accesso al socket solo alle connessioni che arrivano verso tale indirizzo.

Normalmente un client non specifica mai l'indirizzo di un socket, ed il kernel sceglie l'indirizzo di origine quando viene effettuata la connessione, sulla base dell'interfaccia usata per trasmettere i pacchetti, che dipenderà dalle regole di instradamento usate per raggiungere il server (è comunque possibile impostarlo in maniera specifica con i comandi di gestione avanzata del routing, vedi sez. 7.3.4 di [?]). Se un server non specifica il suo indirizzo locale il kernel userà come indirizzo di origine l'indirizzo di destinazione specificato dal SYN del client.

Per specificare un indirizzo generico, con IPv4 si usa il valore `INADDR_ANY`, il cui valore, come accennato in sez. 14.2.2, è pari a zero; nell'esempio fig. 15.9 si è usata un'assegnazione immediata del tipo:

```
serv_add.sin_addr.s_addr = htonl(INADDR_ANY);
```

⁴nel nostro caso la utilizzeremo per socket TCP, ma la funzione è generica e deve essere usata per qualunque tipo di socket `SOCK_STREAM` prima che questo possa accettare connessioni.

⁵un'eccezione a tutto ciò sono i server che usano RPC. In questo caso viene fatta assegnare dal kernel una porta effimera che poi viene registrata presso il *portmapper*; quest'ultimo è un altro demone che deve essere contattato dai client per ottenere la porta effimera su cui si trova il server.

Si noti che si è usato `htonl` per assegnare il valore `INADDR_ANY`, anche se, essendo questo nullo, il riordinamento è inutile. Si tenga presente comunque che tutte le costanti `INADDR_` (riportate in tab. 15.1) sono definite secondo l'*endianness* della macchina, ed anche se esse possono essere invarianti rispetto all'ordinamento dei bit, è comunque buona norma usare sempre la funzione `htonl`.

Costante	Significato
<code>INADDR_ANY</code>	Indirizzo generico (0.0.0.0)
<code>INADDR_BROADCAST</code>	Indirizzo di <i>broadcast</i> .
<code>INADDR_LOOPBACK</code>	Indirizzo di <i>loopback</i> (127.0.0.1).
<code>INADDR_NONE</code>	Indirizzo errato.

Tabella 15.1: Costanti di definizione di alcuni indirizzi generici per IPv4.

L'esempio precedente funziona correttamente con IPv4 poiché che l'indirizzo è rappresentabile anche con un intero a 32 bit; non si può usare lo stesso metodo con IPv6, in cui l'indirizzo deve necessariamente essere specificato con una struttura, perché il linguaggio C non consente l'uso di una struttura costante come operando a destra in una assegnazione.

Per questo motivo nell'header `netinet/in.h` è definita una variabile `in6addr_any` (dichiarata come `extern`, ed inizializzata dal sistema al valore `IN6ADDR_ANY_INIT`) che permette di effettuare una assegnazione del tipo:

```
serv_add.sin6_addr = in6addr_any;
```

in maniera analoga si può utilizzare la variabile `in6addr_loopback` per indicare l'indirizzo di *loopback*, che a sua volta viene inizializzata staticamente a `IN6ADDR_LOOPBACK_INIT`.

15.2.2 La funzione `connect`

La funzione `connect` è usata da un client TCP per stabilire la connessione con un server TCP, ma la funzione è generica e supporta vari tipi di socket. La differenza è che per socket senza connessione come quelli di tipo `SOCK_DGRAM` la sua chiamata si limiterà ad impostare l'indirizzo dal quale e verso il quale saranno inviati e ricevuti i pacchetti, mentre per socket di tipo `SOCK_STREAM` o `SOCK_SEQPACKET` essa attiverà la procedura di avvio della connessione (nel caso del TCP il *three way handshake*). Il suo prototipo è il seguente:

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen)
    Stabilisce una connessione fra due socket.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EACCES, EPERM si è tentato di eseguire una connessione ad un indirizzo *broadcast* senza che il socket fosse stato abilitato per il *broadcast*.

EAFNOSUPPORT l'indirizzo non ha una famiglia di indirizzi corretta nel relativo campo.

EAGAIN non ci sono più porte locali libere.

EALREADY il socket è non bloccante (vedi sez. 10.2.1) e un tentativo precedente di connessione non si è ancora concluso.

ECONNREFUSED non c'è nessuno in ascolto sull'indirizzo remoto.

EINPROGRESS il socket è non bloccante (vedi sez. 10.2.1) e la connessione non può essere conclusa immediatamente.

ENETUNREACH la rete non è raggiungibile.

ETIMEDOUT si è avuto timeout durante il tentativo di connessione.

ed inoltre **EADDRINUSE**, **EBADF**, **EFAULT**, **EINTR**, **EISCONN** e **ENOTSOCK** e nel loro significato generico.

Il primo argomento è un file descriptor ottenuto da una precedente chiamata a `socket`, mentre il secondo e terzo argomento sono rispettivamente l'indirizzo e la dimensione della struttura che contiene l'indirizzo del socket, già descritta in sez. 14.2.

La struttura dell'indirizzo deve essere inizializzata con l'indirizzo IP e il numero di porta del server a cui ci si vuole connettere usando le funzioni illustrate in sez. 14.3 come mostrato nell'esempio che vedremo in sez. 15.3.2.

Nel caso di socket TCP la funzione `connect` avvia il *three way handshake*, e ritorna solo quando la connessione è stabilita o si è verificato un errore. Le possibili cause di errore sono molteplici (ed i relativi codici riportati sopra), quelle che però dipendono dalla situazione della rete e non da errori o problemi nella chiamata della funzione sono le seguenti:

1. Il client non riceve risposta al SYN: l'errore restituito è `ETIMEDOUT`. Stevens riporta che BSD invia un primo SYN alla chiamata di `connect`, un altro dopo 6 secondi, un terzo dopo 24 secondi, se dopo 75 secondi non ha ricevuto risposta viene ritornato l'errore. Linux invece ripete l'emissione del SYN ad intervalli di 30 secondi per un numero di volte che può essere stabilito dall'utente. Questo può essere fatto a livello globale con una opportuna `sysctl` (o più semplicemente scrivendo il valore voluto in `/proc/sys/net/ipv4/tcp_syn_retries`, vedi sez. 16.4.3) e a livello di singolo socket con l'opzione `TCP_SYNCNT` (vedi sez. 16.2.5). Il valore predefinito per la ripetizione dell'invio è di 5 volte, che comporta un timeout dopo circa 180 secondi.
2. Il client riceve come risposta al SYN un RST significa che non c'è nessun programma in ascolto per la connessione sulla porta specificata (il che vuol dire probabilmente che o si è sbagliato il numero della porta o che non è stato avviato il server), questo è un errore fatale e la funzione ritorna non appena il RST viene ricevuto riportando un errore `ECONNREFUSED`.

Il flag RST sta per *reset* ed è un segmento inviato direttamente dal TCP quando qualcosa non va. Tre condizioni che generano un RST sono: quando arriva un SYN per una porta che non ha nessun server in ascolto, quando il TCP abortisce una connessione in corso, quando TCP riceve un segmento per una connessione che non esiste.

3. Il SYN del client provoca l'emissione di un messaggio ICMP di destinazione non raggiungibile. In questo caso dato che il messaggio può essere dovuto ad una condizione

transitoria si ripete l'emissione dei SYN come nel caso precedente, fino al timeout, e solo allora si restituisce il codice di errore dovuto al messaggio ICMP, che da luogo ad un ENETUNREACH.

Se si fa riferimento al diagramma degli stati del TCP riportato in fig. C.1 la funzione `connect` porta un socket dallo stato CLOSED (lo stato iniziale in cui si trova un socket appena creato) prima allo stato SYN_SENT e poi, al ricevimento dell'ACK, nello stato ESTABLISHED. Se invece la connessione fallisce il socket non è più utilizzabile e deve essere chiuso.

Si noti infine che con la funzione `connect` si è specificato solo indirizzo e porta del server, quindi solo una metà della *socket pair*; essendo questa funzione usata nei client l'altra metà contenente indirizzo e porta locale viene lasciata all'assegnazione automatica del kernel, e non è necessario effettuare una `bind`.

15.2.3 La funzione `listen`

La funzione `listen` serve ad usare un socket in modalità passiva, cioè, come dice il nome, per metterlo in ascolto di eventuali connessioni;⁶ in sostanza l'effetto della funzione è di portare il socket dallo stato CLOSED a quello LISTEN. In genere si chiama la funzione in un server dopo le chiamate a `socket` e `bind` e prima della chiamata ad `accept`. Il prototipo della funzione, come definito dalla pagina di manuale, è:

<pre>#include <sys/socket.h> int listen(int sockfd, int backlog)</pre>	<p>Pone un socket in attesa di una connessione.</p>
<p>La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso <code>errno</code> assumerà uno dei valori:</p> <p>EADDRINUSE qualche altro socket sta già usando l'indirizzo.</p> <p>EBADF l'argomento <code>sockfd</code> non è un file descriptor valido.</p> <p>ENOTSOCK l'argomento <code>sockfd</code> non è un socket.</p> <p>EOPNOTSUPP il socket è di un tipo che non supporta questa operazione.</p>	

La funzione pone il socket specificato da `sockfd` in modalità passiva e predispone una coda per le connessioni in arrivo di lunghezza pari a `backlog`. La funzione si può applicare solo a socket di tipo `SOCK_STREAM` o `SOCK_SEQPACKET`. L'argomento `backlog` indica il numero massimo di connessioni pendenti accettate; se esso viene ecceduto il client al momento della richiesta della connessione riceverà un errore di tipo `ECONNREFUSED`, o se il protocollo, come accade nel caso del TCP, supporta la ritrasmissione, la richiesta sarà ignorata in modo che la connessione possa venire ritentata.

Per capire meglio il significato di tutto ciò occorre approfondire la modalità con cui il kernel tratta le connessioni in arrivo. Per ogni socket in ascolto infatti vengono mantenute due code:

1. La coda delle connessioni incomplete (*incomplete connection queue*) che contiene una voce per ciascun socket per il quale è arrivato un SYN ma il *three way handshake* non si è ancora concluso. Questi socket sono tutti nello stato `SYN_RECV`.
2. La coda delle connessioni complete (*complete connection queue*) che contiene una voce per ciascun socket per il quale il *three way handshake* è stato completato ma ancora `accept` non è ritornata. Questi socket sono tutti nello stato `ESTABLISHED`.

⁶questa funzione può essere usata con socket che supportino le connessioni, cioè di tipo `SOCK_STREAM` o `SOCK_SEQPACKET`.

Lo schema di funzionamento è descritto in fig. 15.5: quando arriva un segmento SYN da un client il kernel crea una voce nella coda delle connessioni incomplete e risponde con il segmento SYN+ACK. La voce resterà nella coda delle connessioni incomplete fino al ricevimento del segmento ACK dal client o fino ad un timeout.

Nel caso di completamento del *three way handshake* la voce viene spostata nella coda delle connessioni complete. Quando il processo chiama la funzione `accept` (vedi sez. 15.2.4) gli viene passata la prima voce nella coda delle connessioni complete, oppure, se la coda è vuota, il processo viene posto in attesa in stato di *sleep* e risvegliato all'arrivo della prima connessione completa.

Figura 15.5: Schema di funzionamento delle code delle connessioni complete ed incomplete.

Storicamente il valore dell'argomento `backlog` era corrispondente al massimo valore della somma del numero di voci possibili per ciascuna delle due code. Stevens in [?] riporta che BSD ha sempre applicato un fattore di 1.5 a detto valore, e fornisce una tabella con i risultati ottenuti con vari kernel, compreso anche il vecchio Linux 2.0, che mostrano le differenze fra diverse realizzazioni.

In Linux il significato di questo valore è cambiato a partire dal kernel 2.2 per prevenire il *denial of service* chiamato *SYN flood*. Questo attacco si basa sull'emissione da parte dell'attaccante di un grande numero di segmenti SYN indirizzati verso una porta, forgiati con indirizzo IP fasullo (con la tecnica che viene detta *ip spoofing*); in questo modo i segmenti SYN+ACK di risposta vanno perduti e la coda delle connessioni incomplete viene saturata, impedendo di fatto ulteriori connessioni.

Per ovviare a questo problema il significato del `backlog` è stato cambiato e adesso indica la lunghezza della coda delle connessioni complete. La lunghezza della coda delle connessioni incomplete può essere ancora controllata ma occorre usare esplicitamente la funzione `sysctl` con il parametro `NET_TCP_MAX_SYN_BACKLOG` o scrivere il valore direttamente sul file `/proc/sys/net/ipv4/tcp_max_syn_backlog`.

Quando si attiva la protezione dei *syncookies* però (con l'opzione da compilare nel kernel e da attivare usando `sysctl` o scrivendo nel file `/proc/sys/net/ipv4/tcp_syncookies`, vedi sez. 16.4.3) questo valore viene ignorato e non esiste più un valore massimo. In ogni caso in Linux il valore di `backlog` viene sempre troncato ad un massimo di `SOMAXCONN` se è superiore a detta costante (che di default vale 128); per i kernel precedenti il 2.4.25 questo valore era fisso e non modificabile, nelle versioni successive può essere controllato con un parametro di `sysctl`, o scrivendo nel file `/proc/sys/net/core/somaxconn` (vedi sez. 16.3.3).

La scelta storica per il valore assegnato a questo argomento era di 5, e alcuni vecchi kernel non supportavano neanche valori superiori, ma la situazione corrente è molto cambiata per via della presenza di server web che devono gestire un gran numero di connessioni per cui un tale valore non è più adeguato. Non esiste comunque una risposta univoca per la scelta del valore, per questo non conviene specificarlo con una costante (il cui cambiamento richiederebbe la ricompilazione del server) ma usare piuttosto una variabile di ambiente (vedi sez. 2.3.3).

Stevens tratta accuratamente questo argomento in [?], con esempi presi da casistiche reali trovate su dei server web, ed in particolare evidenzia come non sia più vero che il compito principale della coda sia quello di gestire il caso in cui il server è occupato fra chiamate successive alla `accept` (per cui la coda più occupata sarebbe quella delle connessioni completate), ma piuttosto quello di gestire la presenza di un gran numero di SYN in attesa di concludere il *three way handshake*.

Infine va messo in evidenza che, nel caso di socket TCP, quando un segmento SYN arriva con tutte le code piene, il pacchetto verrà semplicemente ignorato. Questo avviene perché la

condizione in cui le code sono piene è ovviamente transitoria, per cui se il client ritrasmette in seguito un segmento SYN, come previsto dal protocollo, è probabile che essendo passato un po' di tempo esso possa trovare nella coda lo spazio per una nuova connessione.

Se al contrario si rispondesse immediatamente con un segmento RST, per indicare che è impossibile effettuare la connessione, la chiamata a `connect` eseguita dal client fallirebbe ritornando una condizione di errore. In questo modo si sarebbe costretti ad inserire nell'applicazione la gestione dei tentativi di riconnessione, che invece grazie a questa modalità di funzionamento viene effettuata in maniera trasparente dal protocollo TCP.

15.2.4 La funzione `accept`

La funzione `accept` è chiamata da un server per gestire la connessione, nel caso di TCP una volta che sia stato completato il *three way handshake*;⁷ la funzione restituisce un nuovo socket descriptor su cui si potrà operare per effettuare la comunicazione. Se non ci sono connessioni complete il processo viene messo in attesa. Il prototipo della funzione è il seguente:

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

Accetta una connessione sul socket specificato.

La funzione ritorna un numero di socket descriptor positivo in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

EAGAIN o **EWOLDBLOCK** il socket è stato impostato come non bloccante (vedi sez. 10.2.1), e non ci sono connessioni in attesa di essere accettate. In generale possono essere restituiti entrambi i valori, per cui se si ha a cuore la portabilità occorre controllare entrambi.

EBADF l'argomento `sockfd` non è un file descriptor valido.

ECONNABORTED la connessione è stata abortita.

EINTR la funzione è stata interrotta da un segnale.

EINVAL il socket non è in ascolto o `addrlen` non ha un valore valido.

ENOBUFS, **ENOMEM** questo spesso significa che l'allocazione della memoria è limitata dai limiti sui buffer dei socket, non dalla memoria di sistema.

ENOTSOCK l'argomento `sockfd` non è un socket.

EOPNOTSUPP il socket è di un tipo che non supporta questa operazione.

EPERM le regole del firewall non consentono la connessione.

ed inoltre nel loro significato generico: **EFAULT**, **EMFILE**, **ENFILE**; infine a seconda del protocollo e del kernel possono essere restituiti errori di rete relativi al nuovo socket come: **ENOSR**, **ESOCKTNOSUPPORT**, **EPROTONOSUPPORT**, **ETIMEDOUT**, **ERESTARTSYS**.

La funzione estrae la prima connessione relativa al socket `sockfd` in attesa sulla coda delle connessioni complete, che associa ad nuovo socket con le stesse caratteristiche di `sockfd`. Il socket originale non viene toccato e resta nello stato di `LISTEN`, mentre il nuovo socket viene posto nello stato `ESTABLISHED`.

I due argomenti `addr` e `addrlen` (si noti che quest'ultimo è un *value-result argument* passato con un puntatore per riavere indietro il valore) sono usati rispettivamente per ottenere l'indirizzo del client da cui proviene la connessione e la lunghezza dello stesso; la dimensione dipende da quale famiglia di indirizzi si sta utilizzando. Prima della chiamata `addrlen` deve essere inizializzato alle dimensioni della struttura degli indirizzi cui punta `addr` (un numero positivo); al ritorno della funzione `addrlen` conterrà il numero di byte scritti dentro `addr`. Se questa informazione non interessa basterà inizializzare a `NULL` detti puntatori.

⁷come le precedenti, la funzione è generica ed è utilizzabile su socket di tipo `SOCK_STREAM`, `SOCK_SEQPACKET` e `SOCK_RDM`, ma qui la tratteremo solo per gli aspetti riguardanti le connessioni con TCP.

Se la funzione ha successo restituisce il descrittore di un nuovo socket creato dal kernel (detto *connected socket*) a cui viene associata la prima connessione completa (estratta dalla relativa coda, vedi sez. 15.2.3) che il client ha effettuato verso il socket `sockfd`. Quest'ultimo (detto *listening socket*) è quello creato all'inizio e messo in ascolto con `listen`, e non viene toccato dalla funzione. Se non ci sono connessioni pendenti da accettare la funzione mette in attesa il processo⁸ fintanto che non ne arriva una.

La funzione può essere usata solo con socket che supportino la connessione (cioè di tipo `SOCK_STREAM`, `SOCK_SEQPACKET` o `SOCK_RDM`). Per alcuni protocolli che richiedono una conferma esplicita della connessione,⁹ la funzione opera solo l'estrazione dalla coda delle connessioni, la conferma della connessione viene eseguita implicitamente dalla prima chiamata ad una `read` o una `write`, mentre il rifiuto della connessione viene eseguito con la funzione `close`.

Si tenga presente che con Linux, seguendo POSIX.1, è sufficiente includere `sys/socket.h`, ma alcune implementazioni di altri sistemi possono richiedere l'inclusione di `sys/types.h`, per cui dovendo curare la portabilità può essere il caso di includere anche questo file. Inoltre Linux presenta un comportamento diverso nella gestione degli errori rispetto ad altre realizzazioni dei socket BSD, infatti la funzione `accept` passa gli errori di rete pendenti sul nuovo socket come codici di errore per `accept`, per cui l'applicazione deve tenerne conto ed eventualmente ripetere la chiamata alla funzione come per l'errore di `EAGAIN` (torneremo su questo in sez. 15.5).

Un'altra differenza con BSD è che la funzione non fa ereditare al nuovo socket i flag del socket originale, come `O_NONBLOCK`,¹⁰ che devono essere rispecificati ogni volta. Tutto questo deve essere tenuto in conto se si devono scrivere programmi portabili. Per poter effettuare questa impostazione in maniera atomica, senza dover ricorrere ad ulteriori chiamate a `fcntl` su Linux è disponibile anche la funzione `accept4`, il cui prototipo è:¹¹

```
#include <sys/socket.h>
int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags)
    Accetta una connessione sul socket specificato.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà gli stessi valori di `accept`.

La funzione aggiunge un quarto argomento `flags` usato come maschera binaria, e se questo è nullo il suo comportamento è identico a quello di `accept`. Con `flags` si possono impostare contestualmente all'esecuzione sul file descriptor restituito i due flag di `O_NONBLOCK` e `O_CLOEXEC`, fornendo un valore che sia un OR aritmetico delle costanti in tab.15.2.

Costante	Significato
<code>SOCK_NONBLOCK</code>	imposta sul file descriptor restituito il flag di <code>O_NONBLOCK</code>
<code>SOCK_NOEXEC</code>	imposta sul file descriptor restituito il flag di <code>O_CLOEXEC</code>

Tabella 15.2: Costanti per i possibili valori dell'argomento `flags` di `accept4`.

Il meccanismo di funzionamento di `accept` è essenziale per capire il funzionamento di un server: in generale infatti c'è sempre un solo socket in ascolto, detto per questo *listening socket*, che resta per tutto il tempo nello stato `LISTEN`, mentre le connessioni vengono gestite dai nuovi socket, detti *connected socket*, ritornati da `accept`, che si trovano automaticamente nello stato `ESTABLISHED`, e vengono utilizzati per lo scambio dei dati, che avviene su di essi, fino alla chiusura della connessione. Si può riconoscere questo schema anche nell'esempio

⁸a meno che non si sia impostato il socket per essere non bloccante (vedi sez. 10.2.1), nel qual caso ritorna con l'errore `EAGAIN`. Torneremo su questa modalità di operazione in sez. 15.6.

⁹attualmente in Linux solo DECnet ha questo comportamento.

¹⁰ed in generale tutti quelli che si possono impostare con `fcntl`, vedi sez. 5.2.5.

¹¹la funzione è utilizzabile solo se si è definito la macro `_GNU_SOURCE` ed ovviamente non è portabile.

elementare di fig. 15.9, dove per ogni connessione il socket creato da **accept** viene chiuso dopo l'invio dei dati.

15.2.5 Le funzioni **getsockname** e **getpeername**

Oltre a tutte quelle viste finora, dedicate all'utilizzo dei socket, esistono alcune funzioni ausiliarie che possono essere usate per recuperare alcune informazioni relative ai socket ed alle connessioni ad essi associate. Le due più elementari sono le seguenti, usate per ottenere i dati relativi alla *socket pair* associata ad un certo socket. La prima è **getsockname** e serve ad ottenere l'indirizzo locale associato ad un socket; il suo prototipo è:

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *name, socklen_t *namelen)
                                                    Legge l'indirizzo locale di un socket.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso **errno** assumerà uno dei valori:

EBADF l'argomento **sockfd** non è un file descriptor valido.

ENOBUFS non ci sono risorse sufficienti nel sistema per

ENOTSOCK l'argomento **sockfd** non è un socket. eseguire l'operazione.

ed **EFAULT** nel suo significato generico.

La funzione restituisce la struttura degli indirizzi del socket **sockfd** nella struttura indicata dal puntatore **name** la cui lunghezza è specificata tramite l'argomento **namlen**. Quest'ultimo è un *value result argument* e pertanto viene passato come indirizzo per avere indietro anche il numero di byte effettivamente scritti nella struttura puntata da **name**. Si tenga presente che se si è utilizzato un buffer troppo piccolo per **name** l'indirizzo risulterà troncato.

La funzione si usa tutte le volte che si vuole avere l'indirizzo locale di un socket; ad esempio può essere usata da un client (che usualmente non chiama **bind**) per ottenere l'indirizzo IP e la porta locale associati al socket restituito da una **connect**, o da un server che ha chiamato **bind** su un socket usando 0 come porta locale per ottenere il numero di porta effimera assegnato dal kernel.

Inoltre quando un server esegue una **bind** su un indirizzo generico, se chiamata dopo il completamento di una connessione sul socket restituito da **accept**, restituisce l'indirizzo locale che il kernel ha assegnato a quella connessione. Invece tutte le volte che si vuole avere l'indirizzo remoto di un socket si usa la funzione **getpeername**, il cui prototipo è:

```
#include <sys/socket.h>
int getpeername(int sockfd, struct sockaddr * name, socklen_t * namelen)
                                                    Legge l'indirizzo remoto di un socket.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso **errno** assumerà uno dei valori:

EBADF l'argomento **sockfd** non è un file descriptor valido.

ENOBUFS non ci sono risorse sufficienti nel sistema per eseguire l'operazione.

ENOTCONN il socket non è connesso.

ENOTSOCK l'argomento **sockfd** non è un socket.

ed **EFAULT** nel suo significato generico.

La funzione è identica a **getsockname**, ed usa la stessa sintassi, ma restituisce l'indirizzo remoto del socket, cioè quello associato all'altro capo della connessione. Ci si può chiedere a cosa serva questa funzione dato che dal lato client l'indirizzo remoto è sempre noto quando si esegue la **connect** mentre dal lato server si possono usare, come vedremo in fig. 15.10, i valori di ritorno di **accept**.

Il fatto è che in generale quest'ultimo caso non è sempre possibile. In particolare questo avviene quando il server, invece di gestire la connessione direttamente in un processo figlio, come vedremo nell'esempio di server concorrente di sez. 15.3.4, lancia per ciascuna connessione un altro programma, usando `exec`. Questa ad esempio è la modalità con cui opera il *super-server* `xinetd`, che può gestire tutta una serie di servizi diversi, eseguendo su ogni connessione ricevuta sulle porte tenute sotto controllo, il relativo server.

In questo caso benché il processo figlio abbia una immagine della memoria che è copia di quella del processo padre (e contiene quindi anche la struttura ritornata da `accept`), all'esecuzione di `exec` verrà caricata in memoria l'immagine del programma eseguito, che a questo punto perde ogni riferimento ai valori tornati da `accept`. Il socket descriptor però resta aperto, e se si è seguita una opportuna convenzione per rendere noto al programma eseguito qual è il socket connesso (ad esempio il solito `xinetd` fa sempre in modo che i file descriptor 0, 1 e 2 corrispondano al socket connesso) quest'ultimo potrà usare la funzione `getpeername` per determinare l'indirizzo remoto del client.

Infine è da chiarire (si legga la pagina di manuale) che, come per `accept`, il terzo argomento, che è specificato dallo standard POSIX.1g come di tipo `socklen_t *` in realtà deve sempre corrispondere ad un `int *` come prima dello standard, perché tutte le realizzazioni dei socket BSD fanno questa assunzione.

15.2.6 La funzione `close`

La funzione standard Unix `close` (vedi sez. 5.1.2) che si usa sui file può essere usata con lo stesso effetto anche sui file descriptor associati ad un socket.

L'azione di questa funzione quando applicata a socket è di marcarlo come chiuso e ritornare immediatamente al processo. Una volta chiamata il socket descriptor non è più utilizzabile dal processo e non può essere usato come argomento per una `write` o una `read` (anche se l'altro capo della connessione non avesse chiuso la sua parte). Il kernel invierà comunque tutti i dati che ha in coda prima di iniziare la sequenza di chiusura. Vedremo più avanti in sez. 16.2.2 come sia possibile cambiare questo comportamento, e cosa può essere fatto perché il processo possa assicurarsi che l'altro capo abbia ricevuto tutti i dati.

Come per tutti i file descriptor anche per i socket viene mantenuto un numero di riferimenti, per cui se più di un processo ha lo stesso socket aperto l'emissione del FIN e la sequenza di chiusura di TCP non viene innescata fintanto che il numero di riferimenti non si annulla, questo si applica, come visto in sez. 5.2.1, sia ai file descriptor duplicati che a quelli ereditati dagli eventuali processi figli, ed è il comportamento che ci si aspetta in una qualunque applicazione client/server.

Per attivare immediatamente l'emissione del FIN e la sequenza di chiusura descritta in sez. 15.1.3, si può invece usare la funzione `shutdown` su cui torneremo in seguito (vedi sez. 15.6.3).

15.3 Un esempio elementare: il servizio *daytime*

Avendo introdotto le funzioni di base per la gestione dei socket, potremo vedere in questa sezione un primo esempio di applicazione elementare che realizza il servizio *daytime* su TCP, secondo quanto specificato dall'RFC 867. Prima di passare agli esempi del client e del server, inizieremo riesaminando con maggiori dettagli una peculiarità delle funzioni di I/O, già accennata in sez. 5.1.4 e sez. 5.1.5, che nel caso dei socket è particolarmente rilevante. Passeremo poi ad illustrare gli esempi della realizzazione, sia dal lato client, che dal lato server, che si è effettuata sia in forma iterativa che concorrente.

15.3.1 Il comportamento delle funzioni di I/O

Una cosa che si tende a dimenticare quando si ha a che fare con i socket è che le funzioni di input/output non sempre hanno lo stesso comportamento che avrebbero con i normali file di dati (in particolare questo accade per i socket di tipo stream).

Infatti con i socket è comune che funzioni come `read` o `write` possano restituire in input o scrivere in output un numero di byte minore di quello richiesto. Come già accennato in sez. 5.1.4 questo è un comportamento normale per le funzioni di I/O, ma con i normali file di dati il problema si avverte solo in lettura, quando si incontra la fine del file. In generale non è così, e con i socket questo è particolarmente evidente.

```

1 #include <unistd.h>
2
3 ssize_t FullRead(int fd, void *buf, size_t count)
4 {
5     size_t nleft;
6     ssize_t nread;
7
8     nleft = count;
9     while (nleft > 0) {                /* repeat until no left */
10         if ( (nread = read(fd, buf, nleft)) < 0) {
11             if (errno == EINTR) {      /* if interrupted by system call */
12                 continue;             /* repeat the loop */
13             } else {
14                 return(nread);         /* otherwise exit */
15             }
16         } else if (nread == 0) {        /* EOF */
17             break;                     /* break loop here */
18         }
19         nleft -= nread;                 /* set left to read */
20         buf += nread;                  /* set pointer */
21     }
22     return (nleft);
23 }

```

Figura 15.6: La funzione `FullRead`, che legge esattamente `count` byte da un file descriptor, iterando opportunamente le letture.

Quando ci si trova ad affrontare questo comportamento tutto quello che si deve fare è semplicemente ripetere la lettura (o la scrittura) per la quantità di byte restanti, tenendo conto che le funzioni si possono bloccare se i dati non sono disponibili: è lo stesso comportamento che si può avere scrivendo più di `PIPE_BUF` byte in una *pipe* (si riveda quanto detto in sez. 11.1.1).

Per questo motivo, seguendo l'esempio di R. W. Stevens in [?], si sono definite due funzioni, `FullRead` e `FullWrite`, che eseguono lettura e scrittura tenendo conto di questa caratteristica, ed in grado di ritornare solo dopo avere letto o scritto esattamente il numero di byte specificato; il sorgente è riportato rispettivamente in fig. 15.6 e fig. 15.7 ed è disponibile fra i sorgenti allegati alla guida nei file `FullRead.c` e `FullWrite.c`.

Come si può notare le due funzioni ripetono la lettura/scrittura in un ciclo fino all'esaurimento del numero di byte richiesti, in caso di errore viene controllato se questo è `EINTR` (cioè un'interruzione della *system call* dovuta ad un segnale), nel qual caso l'accesso viene ripetuto, altrimenti l'errore viene ritornato al programma chiamante, interrompendo il ciclo.

Nel caso della lettura, se il numero di byte letti è zero, significa che si è arrivati alla fine del file (per i socket questo significa in genere che l'altro capo è stato chiuso, e quindi non sarà

```

1 #include <unistd.h>
2
3 ssize_t FullWrite(int fd, const void *buf, size_t count)
4 {
5     size_t nleft;
6     ssize_t nwritten;
7
8     nleft = count;
9     while (nleft > 0) {          /* repeat until no left */
10         if ( (nwritten = write(fd, buf, nleft)) < 0) {
11             if (errno == EINTR) { /* if interrupted by system call */
12                 continue;        /* repeat the loop */
13             } else {
14                 return(nwritten); /* otherwise exit with error */
15             }
16         }
17         nleft -= nwritten;        /* set left to write */
18         buf +=nwritten;          /* set pointer */
19     }
20     return (nleft);
21 }

```

Figura 15.7: La funzione `FullWrite`, che scrive esattamente `count` byte su un file descriptor, iterando opportunamente le scritture.

più possibile leggere niente) e pertanto si ritorna senza aver concluso la lettura di tutti i byte richiesti. Entrambe le funzioni restituiscono 0 in caso di successo, ed un valore negativo in caso di errore, `FullRead` restituisce il numero di byte non letti in caso di *end-of-file* prematuro.

15.3.2 Il client *daytime*

Il primo esempio di applicazione delle funzioni di base illustrate in sez. 15.2 è relativo alla creazione di un client elementare per il servizio *daytime*, un servizio elementare, definito nell’RFC 867, che restituisce l’ora locale della macchina a cui si effettua la richiesta, e che è assegnato alla porta 13.

In fig. 15.8 è riportata la sezione principale del codice del nostro client. Il sorgente completo del programma (`TCP_daytime.c`, che comprende il trattamento delle opzioni ed una funzione per stampare un messaggio di aiuto) è allegato alla guida nella sezione dei codici sorgente e può essere compilato su una qualunque macchina GNU/Linux.

Il programma anzitutto (1–5) include gli header necessari; dopo la dichiarazione delle variabili (9–12) si è omessa tutta la parte relativa al trattamento degli argomenti passati dalla linea di comando (effettuata con le apposite funzioni illustrate in sez. 2.3.2).

Il primo passo (14–18) è creare un socket TCP (quindi di tipo `SOCK_STREAM` e di famiglia `AF_INET`). La funzione `socket` ritorna il descrittore che viene usato per identificare il socket in tutte le chiamate successive. Nel caso la chiamata fallisca si stampa un errore (16) con la funzione `perror` e si esce (17) con un codice di errore.

Il passo seguente (19–27) è quello di costruire un’apposita struttura `sockaddr_in` in cui sarà inserito l’indirizzo del server ed il numero della porta del servizio. Il primo passo (20) è inizializzare tutto a zero, per poi inserire il tipo di indirizzo (21) e la porta (22), usando per quest’ultima la funzione `htons` per convertire il formato dell’intero usato dal computer a

```

1 #include <stdlib.h>    /* predefined types */
2 #include <unistd.h>    /* include unix standard library */
3 #include <arpa/inet.h> /* IP addresses conversion utilities */
4 #include <sys/socket.h> /* socket library */
5 #include <stdio.h>     /* include standard I/O library */
6 #include <string.h>    /* C strings library */
7
8 int main(int argc, char *argv[])
9 {
10     int sock_fd;
11     int i, nread;
12     struct sockaddr_in serv_add;
13     char buffer[MAXLINE];
14     ...
15     /* create socket */
16     if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
17         perror("Socket_creation_error");
18         return -1;
19     }
20     /* initialize address */
21     memset((void *) &serv_add, 0, sizeof(serv_add)); /* clear server address */
22     serv_add.sin_family = AF_INET; /* address type is INET */
23     serv_add.sin_port = htons(13); /* daytime post is 13 */
24     /* build address using inet_pton */
25     if ( (inet_pton(AF_INET, argv[optind], &serv_add.sin_addr)) <= 0) {
26         perror("Address_creation_error");
27         return -1;
28     }
29     /* establish connection */
30     if (connect(sock_fd, (struct sockaddr *)&serv_add, sizeof(serv_add)) < 0) {
31         perror("Connection_error");
32         return -1;
33     }
34     /* read daytime from server */
35     while ( (nread = read(sock_fd, buffer, MAXLINE)) > 0) {
36         buffer[nread]=0;
37         if (fputs(buffer, stdout) == EOF) { /* write daytime */
38             perror("fputs_error");
39             return -1;
40         }
41     }
42     /* error on read */
43     if (nread < 0) {
44         perror("Read_error");
45         return -1;
46     }
47     /* normal exit */
48     return 0;
49 }

```

Figura 15.8: Esempio di codice di un client elementare per il servizio *daytime*.

quello usato nella rete, infine (23-27) si può utilizzare la funzione `inet_pton` per convertire l'indirizzo numerico passato dalla linea di comando.

A questo punto (28-32) usando la funzione `connect` sul socket creato in precedenza (29) si può stabilire la connessione con il server. Per questo si deve utilizzare come secondo argomento la struttura preparata in precedenza con il relativo indirizzo; si noti come, esistendo diversi tipi di socket, si sia dovuto effettuare un cast. Un valore di ritorno della funzione negativo implica il fallimento della connessione, nel qual caso si stampa un errore (30) e si ritorna (31).

Completata con successo la connessione il passo successivo (34-40) è leggere la data dal socket; il protocollo prevede che il server invii sempre una stringa alfanumerica, il formato della stringa non è specificato dallo standard, per cui noi useremo il formato usato dalla funzione `ctime`, seguito dai caratteri di terminazione `\r\n`, cioè qualcosa del tipo:

```
Wed Apr 4 00:53:00 2001\r\n
```

questa viene letta dal socket (34) con la funzione `read` in un buffer temporaneo; la stringa poi deve essere terminata (35) con il solito carattere nullo per poter essere stampata (36) sullo *standard output* con l'uso di `fputs`.

Come si è già spiegato in sez. 15.3.1 la risposta dal socket potrà arrivare in un unico pacchetto di 26 byte (come avverrà senz'altro nel caso in questione) ma potrebbe anche arrivare in 26 pacchetti di un byte. Per questo nel caso generale non si può mai assumere che tutti i dati arrivino con una singola lettura, pertanto quest'ultima deve essere effettuata in un ciclo in cui si continui a leggere fintanto che la funzione `read` non ritorni uno zero (che significa che l'altro capo ha chiuso la connessione) o un numero minore di zero (che significa un errore nella connessione).

Si noti come in questo caso la fine dei dati sia specificata dal server che chiude la connessione (anche questo è quanto richiesto dal protocollo); questa è una delle tecniche possibili (è quella usata pure dal protocollo HTTP), ma ce ne possono essere altre, ad esempio FTP marca la conclusione di un blocco di dati con la sequenza ASCII `\r\n` (carriage return e line feed), mentre il DNS mette la lunghezza in testa ad ogni blocco che trasmette. Il punto essenziale è che TCP non provvede nessuna indicazione che permetta di marcare dei blocchi di dati, per cui se questo è necessario deve provvedere il programma stesso.

Se abilitiamo il servizio *daytime*¹² possiamo verificare il funzionamento del nostro client, avremo allora:

```
[piccardi@gont sources]$ ./daytime 127.0.0.1
Mon Apr 21 20:46:11 2003
```

e come si vede tutto funziona regolarmente.

15.3.3 Un server *daytime* iterativo

Dopo aver illustrato il client daremo anche un esempio di un server elementare, che sia anche in grado di rispondere al precedente client. Come primo esempio realizzeremo un server iterativo, in grado di fornire una sola risposta alla volta. Il codice del programma è nuovamente mostrato in fig. 15.9, il sorgente completo (`TCP_iter_daytimed.c`) è allegato insieme agli altri file degli esempi.

Come per il client si includono (1-9) gli header necessari a cui è aggiunto quello per trattare i tempi, e si definiscono (14-18) alcune costanti e le variabili necessarie in seguito.

¹²in genere questo viene fornito direttamente dal *superdemon* `xinetd`, pertanto basta assicurarsi che esso sia abilitato nel relativo file di configurazione.

```

1 #define MAXLINE 80
2 #define BACKLOG 10
3 int main(int argc, char *argv[])
4 {
5     /*
6      * Variables definition
7      */
8     int list_fd, conn_fd;
9     int i;
10    struct sockaddr_in serv_add;
11    char buffer[MAXLINE];
12    time_t timeval;
13    ...
14    /* create socket */
15    if ( (list_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
16        perror("Socket_creation_error");
17        exit(-1);
18    }
19    /* initialize address */
20    memset((void *)&serv_add, 0, sizeof(serv_add)); /* clear server address */
21    serv_add.sin_family = AF_INET; /* address type is INET */
22    serv_add.sin_port = htons(13); /* daytime port is 13 */
23    serv_add.sin_addr.s_addr = htonl(INADDR_ANY); /* connect from anywhere */
24    /* bind socket */
25    if (bind(list_fd, (struct sockaddr *)&serv_add, sizeof(serv_add)) < 0) {
26        perror("bind_error");
27        exit(-1);
28    }
29    /* listen on socket */
30    if (listen(list_fd, BACKLOG) < 0 ) {
31        perror("listen_error");
32        exit(-1);
33    }
34    /* write daytime to client */
35    while (1) {
36        if ( (conn_fd = accept(list_fd, (struct sockaddr *) NULL, NULL)) < 0 ) {
37            perror("accept_error");
38            exit(-1);
39        }
40        timeval = time(NULL);
41        snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));
42        if ( (write(conn_fd, buffer, strlen(buffer))) < 0 ) {
43            perror("write_error");
44            exit(-1);
45        }
46        close(conn_fd);
47    }
48    /* normal exit */
49    exit(0);
50 }

```

Figura 15.9: Esempio di codice di un semplice server per il servizio daytime.

Come nel caso precedente si sono omesse le parti relative al trattamento delle opzioni da riga di comando.

La creazione del socket (20-24) è analoga al caso precedente, come pure l'inizializzazione (25-29) della struttura `sockaddr_in`. Anche in questo caso (28) si usa la porta standard del servizio *daytime*, ma come indirizzo IP si usa (29) il valore predefinito `INET_ANY`, che corrisponde all'indirizzo generico.

Si effettua poi (30-34) la chiamata alla funzione `bind` che permette di associare la precedente struttura al socket, in modo che quest'ultimo possa essere usato per accettare connessioni su una qualunque delle interfacce di rete locali. In caso di errore si stampa (31) un messaggio, e si termina (32) immediatamente il programma.

Il passo successivo (35-39) è quello di mettere “*in ascolto*” il socket; questo viene fatto (36) con la funzione `listen` che dice al kernel di accettare connessioni per il socket che abbiamo creato; la funzione indica inoltre, con il secondo argomento, il numero massimo di connessioni che il kernel accetterà di mettere in coda per il suddetto socket. Di nuovo in caso di errore si stampa (37) un messaggio, e si esce (38) immediatamente.

La chiamata a `listen` completa la preparazione del socket per l'ascolto (che viene chiamato anche *listening descriptor*) a questo punto si può procedere con il ciclo principale (40-53) che viene eseguito indefinitamente. Il primo passo (42) è porsi in attesa di connessioni con la chiamata alla funzione `accept`, come in precedenza in caso di errore si stampa (43) un messaggio, e si esce (44).

Il processo resterà in stato di *sleep* fin quando non arriva e viene accettata una connessione da un client; quando questo avviene `accept` ritorna, restituendo un secondo descrittore, che viene chiamato *connected descriptor*, e che è quello che verrà usato dalla successiva chiamata alla `write` per scrivere la risposta al client.

Il ciclo quindi proseguirà determinando (46) il tempo corrente con una chiamata a `time`, con il quale si potrà opportunamente costruire (47) la stringa con la data da trasmettere (48) con la chiamata a `write`. Completata la trasmissione il nuovo socket viene chiuso (52). A questo punto il ciclo si chiude ricominciando da capo in modo da poter ripetere l'invio della data in risposta ad una successiva connessione.

È importante notare che questo server è estremamente elementare, infatti, a parte il fatto di poter essere usato solo con indirizzi IPv4, esso è in grado di rispondere ad un solo client alla volta: è cioè, come dicevamo, un *server iterativo*. Inoltre è scritto per essere lanciato da linea di comando, se lo si volesse utilizzare come demone occorrerebbero le opportune modifiche (come una chiamata a `daemon` prima dell'inizio del ciclo principale) per tener conto di quanto illustrato in sez. 8.1.5. Si noti anche che non si è inserita nessuna forma di gestione della terminazione del processo, dato che tutti i file descriptor vengono chiusi automaticamente alla sua uscita, e che, non generando figli, non è necessario preoccuparsi di gestire la loro terminazione.

15.3.4 Un server *daytime* concorrente

Il server *daytime* dell'esempio in sez. 15.3.3 è un tipico esempio di server iterativo, in cui viene servita una richiesta alla volta; in generale però, specie se il servizio è più complesso e comporta uno scambio di dati più sostanzioso di quello in questione, non è opportuno bloccare un server nel servizio di un client per volta; per questo si ricorre alle capacità di multitasking del sistema.

Come spiegato in sez. 3.1.3 una delle modalità più comuni di funzionamento da parte dei server è quella di usare la funzione `fork` per creare, ad ogni richiesta da parte di un client, un processo figlio che si incarichi della gestione della comunicazione. Si è allora riscritto il server

daytime dell'esempio precedente in forma concorrente, inserendo anche una opzione per la stampa degli indirizzi delle connessioni ricevute.

In fig. 15.10 è mostrato un estratto del codice, in cui si sono tralasciati il trattamento delle opzioni e le parti rimaste invariate rispetto al precedente esempio (cioè tutta la parte riguardante l'apertura passiva del socket). Al solito il sorgente completo del server, nel file `TCP_cunc_daytimed.c`, è allegato insieme ai sorgenti degli altri esempi.

Stavolta (21-26) la funzione `accept` è chiamata fornendo una struttura di indirizzi in cui saranno ritornati l'indirizzo IP e la porta da cui il client effettua la connessione, che in un secondo tempo, (40-44), se il logging è abilitato, stamperemo sullo *standard output*.

Quando `accept` ritorna il server chiama la funzione `fork` (27-31) per creare il processo figlio che effettuerà (32-46) tutte le operazioni relative a quella connessione, mentre il padre proseguirà l'esecuzione del ciclo principale in attesa di ulteriori connessioni.

Si noti come il figlio operi solo sul socket connesso, chiudendo immediatamente (33) il socket `list_fd`; mentre il padre continua ad operare solo sul socket in ascolto chiudendo (48) `conn_fd` al ritorno dalla `fork`. Per quanto abbiamo detto in sez. 15.2.6 nessuna delle due chiamate a `close` causa l'innescò della sequenza di chiusura perché il numero di riferimenti al file descriptor non si è annullato.

Infatti subito dopo la creazione del socket `list_fd` ha una referenza, e lo stesso vale per `conn_fd` dopo il ritorno di `accept`, ma dopo la `fork` i descrittori vengono duplicati nel padre e nel figlio per cui entrambi i socket si trovano con due referenze. Questo fa sì che quando il padre chiude `sock_fd` esso resta con una referenza da parte del figlio, e sarà definitivamente chiuso solo quando quest'ultimo, dopo aver completato le sue operazioni, chiamerà (45) la funzione `close`.

In realtà per il figlio non sarebbe necessaria nessuna chiamata a `close`, in quanto con la `exit` finale (45) tutti i file descriptor, quindi anche quelli associati ai socket, vengono automaticamente chiusi. Tuttavia si è preferito effettuare esplicitamente le chiusure per avere una maggiore chiarezza del codice, e per evitare eventuali errori, prevenendo ad esempio un uso involontario del *listening descriptor*.

Si noti invece come sia essenziale che il padre chiuda ogni volta il socket connesso dopo la `fork`; se così non fosse nessuno di questi socket sarebbe effettivamente chiuso dato che alla chiusura da parte del figlio resterebbe ancora un riferimento nel padre. Si avrebbero così due effetti: il padre potrebbe esaurire i descrittori disponibili (che sono un numero limitato per ogni processo) e soprattutto nessuna delle connessioni con i client verrebbe chiusa.

Come per ogni server iterativo il lavoro di risposta viene eseguito interamente dal processo figlio. Questo si incarica (34) di chiamare `time` per leggere il tempo corrente, e di stamparlo (35) sulla stringa contenuta in `buffer` con l'uso di `snprintf` e `ctime`. Poi la stringa viene scritta (36-39) sul socket, controllando che non ci siano errori. Anche in questo caso si è evitato il ricorso a `FullWrite` in quanto la stringa è estremamente breve e verrà senz'altro scritta in un singolo segmento.

Inoltre nel caso sia stato abilitato il *logging* delle connessioni, si provvede anche (40-43) a stampare sullo *standard output* l'indirizzo e la porta da cui il client ha effettuato la connessione, usando i valori contenuti nelle strutture restituite da `accept`, eseguendo le opportune conversioni con `inet_ntop` e `ntohs`.

Ancora una volta l'esempio è estremamente semplificato, si noti come di nuovo non si sia gestita né la terminazione del processo né il suo uso come demone, che tra l'altro sarebbe stato incompatibile con l'uso della opzione di logging che stampa gli indirizzi delle connessioni sullo *standard output*. Un altro aspetto tralasciato è la gestione della terminazione dei processi figli, torneremo su questo più avanti quando tratteremo alcuni esempi di server più complessi.

```

1 #include <sys/types.h>    /* predefined types */
2 #include <unistd.h>       /* include unix standard library */
3 #include <arpa/inet.h>    /* IP addresses conversion utililites */
4 #include <sys/socket.h>   /* socket library */
5 #include <stdio.h>        /* include standard I/O library */
6 #include <time.h>
7
8 int main(int argc, char *argv[])
9 {
10     int list_fd, conn_fd;
11     int i;
12     struct sockaddr_in serv_addr, client;
13     char buffer[MAXLINE];
14     socklen_t len;
15     time_t timeval;
16     pid_t pid;
17     int logging=0;
18     ...
19     /* write daytime to client */
20     while (1) {
21         len = sizeof(client);
22         if ( (conn_fd = accept(list_fd, (struct sockaddr *)&client, &len))
23             < 0 ) {
24             perror("accept_error");
25             exit(-1);
26         }
27         /* fork to handle connection */
28         if ( (pid = fork()) < 0 ){
29             perror("fork_error");
30             exit(-1);
31         }
32         if (pid == 0) {                /* child */
33             close(list_fd);
34             timeval = time(NULL);
35             snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));
36             if ( (write(conn_fd, buffer, strlen(buffer))) < 0 ) {
37                 perror("write_error");
38                 exit(-1);
39             }
40             if (logging) {
41                 inet_ntop(AF_INET, &client.sin_addr, buffer, sizeof(buffer));
42                 printf("Request_from_host_%s,_port_%d\n", buffer,
43                     ntohs(client.sin_port));
44             }
45             close(conn_fd);
46             exit(0);
47         } else {                      /* parent */
48             close(conn_fd);
49         }
50     }
51     /* normal exit, never reached */
52     exit(0);
53 }

```

Figura 15.10: Esempio di codice di un server concorrente elementare per il servizio daytime.

15.4 Un esempio più completo: il servizio *echo*

L'esempio precedente, basato sul servizio *daytime*, è un esempio molto elementare, in cui il flusso dei dati va solo nella direzione dal server al client. In questa sezione esamineremo un esempio di applicazione client/server un po' più complessa, che usi i socket TCP per una comunicazione in entrambe le direzioni.

Ci limiteremo a fornire una realizzazione elementare, che usi solo le funzioni di base viste finora, ma prenderemo in esame, oltre al comportamento in condizioni normali, anche tutti i possibili scenari particolari (errori, sconnessione della rete, crash del client o del server durante la connessione) che possono avere luogo durante l'impiego di un'applicazione di rete, partendo da una versione primitiva che dovrà essere rimaneggiata di volta in volta per poter tenere conto di tutte le evenienze che si possono manifestare nella vita reale di un'applicazione di rete, fino ad arrivare ad una realizzazione completa.

15.4.1 Il servizio *echo*

Nella ricerca di un servizio che potesse fare da esempio per una comunicazione bidirezionale, si è deciso, seguendo la scelta di Stevens in [?], di usare il servizio *echo*, che si limita a restituire in uscita quanto immesso in ingresso. Infatti, nonostante la sua estrema semplicità, questo servizio costituisce il prototipo ideale per una generica applicazione di rete in cui un server risponde alle richieste di un client. Nel caso di una applicazione più complessa quello che si potrà avere in più è una elaborazione dell'input del client, che in molti casi viene interpretato come un comando, da parte di un server che risponde fornendo altri dati in uscita.

Il servizio *echo* è uno dei servizi standard solitamente provvisti direttamente dal superserver *inetd*, ed è definito dall'RFC 862. Come dice il nome il servizio deve riscrivere indietro sul socket i dati che gli vengono inviati in ingresso. L'RFC descrive le specifiche del servizio sia per TCP che UDP, e per qìl primo stabilisce che una volta stabilita la connessione ogni dato in ingresso deve essere rimandato in uscita fintanto che il chiamante non ha chiuso la connessione. Al servizio è assegnata la porta riservata 7.

Nel nostro caso l'esempio sarà costituito da un client che legge una linea di caratteri dallo *standard input* e la scrive sul server. A sua volta il server leggerà la linea dalla connessione e la riscriverà immutata all'indietro. Sarà compito del client leggere la risposta del server e stamparla sullo *standard output*.

15.4.2 Il client *echo*: prima versione

Il codice della prima versione del client per il servizio *echo* (file *TCP_echo_first.c*) dei sorgenti allegati alla guida) è riportato in fig. 15.11. Esso ricalca la struttura del precedente client per il servizio *daytime* (vedi sez. 15.3.2), e la prima parte (10-27) è sostanzialmente identica, a parte l'uso di una porta diversa.

Al solito si è tralasciata la sezione relativa alla gestione delle opzioni a riga di comando. Una volta dichiarate le variabili, si prosegue (10-13) con la creazione del socket con l'usuale controllo degli errori, alla preparazione (14-17) della struttura dell'indirizzo, che stavolta usa la porta 7 riservata al servizio *echo*, infine si converte (18-22) l'indirizzo specificato a riga di comando. A questo punto (23-27) si può eseguire la connessione al server secondo la stessa modalità usata in sez. 15.3.2.

Completata la connessione, per gestire il funzionamento del protocollo si usa la funzione *ClientEcho*, il cui codice si è riportato a parte in fig. 15.12. Questa si preoccupa di gestire tutta la comunicazione, leggendo una riga alla volta dallo *standard input* *stdin*, scrivendola

```

1 int main(int argc, char *argv[])
2 {
3     /*
4     * Variables definition
5     */
6     int sock_fd, i;
7     struct sockaddr_in serv_add;
8     ...
9     /* create socket */
10    if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
11        perror("Socket_creation_error");
12        return 1;
13    }
14    /* initialize address */
15    memset((void *) &serv_add, 0, sizeof(serv_add)); /* clear server address */
16    serv_add.sin_family = AF_INET; /* address type is INET */
17    serv_add.sin_port = htons(7); /* echo port is 7 */
18    /* build address using inet_pton */
19    if ( (inet_pton(AF_INET, argv[optind], &serv_add.sin_addr)) <= 0) {
20        perror("Address_creation_error");
21        return 1;
22    }
23    /* establish connection */
24    if (connect(sock_fd, (struct sockaddr *)&serv_add, sizeof(serv_add)) < 0) {
25        perror("Connection_error");
26        return 1;
27    }
28    /* read daytime from server */
29    ClientEcho(stdin, sock_fd);
30    /* normal exit */
31    return 0;
32 }

```

Figura 15.11: Codice della prima versione del client *echo*.

sul socket e ristampando su `stdout` quanto ricevuto in risposta dal server. Al ritorno dalla funzione (30-31) anche il programma termina.

La funzione `ClientEcho` utilizza due buffer (3) per gestire i dati inviati e letti sul socket. La comunicazione viene gestita all'interno di un ciclo (5-10), i dati da inviare sulla connessione vengono presi dallo `stdin` usando la funzione `fgets`, che legge una linea di testo (terminata da un CR e fino al massimo di `MAXLINE` caratteri) e la salva sul buffer di invio.

Si usa poi (6) la funzione `FullWrite`, vista in sez. 15.3.1, per scrivere i dati sul socket, gestendo automaticamente l'invio multiplo qualora una singola `write` non sia sufficiente. I dati vengono riletto indietro (7) con una `read`¹³ sul buffer di ricezione e viene inserita (8) la terminazione della stringa e per poter usare (9) la funzione `fputs` per scriverli su `stdout`.

Quando si concluderà l'invio di dati mandando un *end-of-file* sullo *standard input* si avrà il ritorno di `fgets` con un puntatore nullo (si riveda quanto spiegato in sez. 5.3.5) e la conseguente uscita dal ciclo; al che la subroutine ritorna ed il nostro programma client termina.

Si può effettuare una verifica del funzionamento del client abilitando il servizio *echo* nella configurazione di `xinetd` sulla propria macchina ed usandolo direttamente verso di esso in

¹³Si è fatta l'assunzione implicita che i dati siano contenuti tutti in un solo segmento, così che la chiamata a `read` li restituisca sempre tutti; avendo scelto una dimensione ridotta per il buffer questo sarà sempre vero, vedremo più avanti come superare il problema di rileggere indietro tutti e soli i dati disponibili, senza bloccarsi.

```

1 void ClientEcho(FILE * filein, int socket)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread;
5     while (fgets(sendbuff, MAXLINE, filein) != NULL) {
6         FullWrite(socket, sendbuff, strlen(sendbuff));
7         nread = read(socket, recvbuff, strlen(sendbuff));
8         recvbuff[nread] = 0;
9         fputs(recvbuff, stdout);
10    }
11    return;
12 }

```

Figura 15.12: Codice della prima versione della funzione `ClientEcho` per la gestione del servizio *echo*.

locale, vedremo in dettaglio più avanti (in sez. 15.4.4) il funzionamento del programma, usato però con la nostra versione del server *echo*, che illustriamo immediatamente.

15.4.3 Il server *echo*: prima versione

La prima versione del server, contenuta nel file `TCP_echod_first.c`, è riportata in fig. 15.13 e fig. 15.14. Come abbiamo fatto per il client anche il server è stato diviso in un corpo principale, costituito dalla funzione `main`, che è molto simile a quello visto nel precedente esempio per il server del servizio *daytime* di sez. 15.3.4, e da una funzione ausiliaria `ServEcho` che si cura della gestione del servizio.

In questo caso però, rispetto a quanto visto nell'esempio di fig. 15.10 si è preferito scrivere il server curando maggiormente alcuni dettagli, per tenere conto anche di alcune esigenze generali (che non riguardano direttamente la rete), come la possibilità di lanciare il server anche in modalità interattiva e la cessione dei privilegi di amministratore non appena questi non sono più necessari.

La sezione iniziale del programma (8–21) è la stessa del server di sez. 15.3.4, ed ivi descritta in dettaglio: crea il socket, inizializza l'indirizzo e esegue `bind`; dato che quest'ultima funzione viene usata su una porta riservata, il server dovrà essere eseguito da un processo con i privilegi di amministratore, pena il fallimento della chiamata.

Una volta eseguita la funzione `bind` però i privilegi di amministratore non sono più necessari, per questo è sempre opportuno rilasciarli, in modo da evitare problemi in caso di eventuali vulnerabilità del server. Per questo prima (22–26) si esegue `setgid` per assegnare il processo ad un gruppo senza privilegi,¹⁴ e poi si ripete (27–30) l'operazione usando `setuid` per cambiare anche l'utente.¹⁵ Infine (30–36), qualora sia impostata la variabile `demonize`, prima (31) si apre il sistema di logging per la stampa degli errori, e poi (32–35) si invoca `daemon` per eseguire in background il processo come demone.

A questo punto il programma prosegue nel ciclo principale, illustrato in fig. 15.14, usando lo schema già visto in precedenza per server per il servizio *daytime*, con l'unica differenza della

¹⁴si è usato il valore 65534, ovvero -1 per il formato `short`, che di norma in tutte le distribuzioni viene usato per identificare il gruppo `nogroup` e l'utente `nobody`, usati appunto per eseguire programmi che non richiedono nessun privilegio particolare.

¹⁵si tenga presente che l'ordine in cui si eseguono queste due operazioni è importante, infatti solo avendo i privilegi di amministratore si può cambiare il gruppo di un processo ad un altro di cui non si fa parte, per cui chiamare prima `setuid` farebbe fallire una successiva chiamata a `setgid`. Inoltre si ricordi (si riveda quanto esposto in sez. 3.2) che usando queste due funzioni il rilascio dei privilegi è irreversibile.

```

1 int main(int argc, char *argv[])
2 {
3     int list_fd, conn_fd;
4     pid_t pid;
5     struct sockaddr_in serv_add;
6     ...
7     /* create and init socket */
8     if ( (list_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
9         perror("Socket_creation_error");
10        exit(1);
11    }
12    memset((void *)&serv_add, 0, sizeof(serv_add)); /* clear server address */
13    serv_add.sin_family = AF_INET; /* address type is INET */
14    serv_add.sin_port = htons(7); /* echo port is 7 */
15    serv_add.sin_addr.s_addr = htonl(INADDR_ANY); /* connect from anywhere */
16    if (bind(list_fd, (struct sockaddr *)&serv_add, sizeof(serv_add)) < 0) {
17        perror("bind_error");
18        exit(1);
19    }
20    /* give away privileges and go daemon */
21    if (setgid(65534) != 0) { /* first give away group privileges */
22        perror("cannot_give_away_group_privileges");
23        exit(1);
24    }
25    if (setuid(65534) != 0) { /* and only after user ... */
26        perror("cannot_give_away_user_privileges");
27        exit(1);
28    }
29    if (daemonize) { /* go daemon */
30        openlog(argv[0], 0, LOG_DAEMON); /* open logging */
31        if (daemon(0, 0) != 0) {
32            perror("cannot_start_as_daemon");
33            exit(1);
34        }
35    }
36    ...

```

Figura 15.13: Codice di inizializzazione della prima versione del server per il servizio echo.

chiamata alla funzione `PrintErr`, riportata in fig. 15.15, al posto di `perror` per la stampa degli errori.

Si inizia con il porre (3-6) in ascolto il socket, e poi si esegue indefinitamente il ciclo principale (7-26). All'interno di questo si ricevono (9-12) le connessioni, creando (13-16) un processo figlio per ciascuna di esse. Quest'ultimo (17-21), chiuso (18) il *listening socket*, esegue (19) la funzione di gestione del servizio `ServEcho`, ed al ritorno di questa esce (20).

Il padre invece si limita (22) a chiudere il *connected socket* per ricominciare da capo il ciclo in attesa di nuove connessioni. In questo modo si ha un server concorrente. La terminazione del padre non è gestita esplicitamente, e deve essere effettuata inviando un segnale al processo.

Avendo trattato direttamente la gestione del programma come demone, si è dovuto anche provvedere alla necessità di poter stampare eventuali messaggi di errore attraverso il sistema del *syslog* trattato in sez. 8.1.5. Come accennato questo è stato fatto utilizzando come *wrapper* la funzione `PrintErr`, il cui codice è riportato in fig. 15.15.

In essa ci si limita a controllare (2) se è stato impostato (valore attivo per default) l'uso come demone, nel qual caso (3) si usa *syslog* (vedi sez. 8.1.5) per stampare il messaggio

```

1  ...
2  /* main body */
3  if (listen(list_fd, BACKLOG) < 0) { /* listen on socket */
4      PrintErr("listen_error");
5      exit(1);
6  }
7  while (1) {                                /* handle echo to client */
8      len = sizeof(cli_add);
9      if ( (conn_fd = accept(list_fd, NULL, NULL)) < 0) {
10         PrintErr("accept_error");
11         exit(1);
12     }
13     if ( (pid = fork()) < 0 ) {              /* fork to handle connection */
14         PrintErr("fork_error");
15         exit(1);
16     }
17     if (pid == 0) {                          /* child */
18         close(list_fd);                      /* close listening socket */
19         ServEcho(conn_fd);                  /* handle echo */
20         exit(0);
21     } else {                                /* parent */
22         close(conn_fd);                      /* close connected socket */
23     }
24 }
25 exit(0);    /* normal exit, never reached */
26 }

```

Figura 15.14: Codice di inizializzazione della prima versione del server per il servizio *echo*.

```

1 void PrintErr(char * error) {
2     if (daemonize) {                        /* daemon mode */
3         syslog(LOG_ERR, "%s: %m", error); /* log string and error message */
4     } else {
5         perror(error);
6     }
7     return;
8 }

```

Figura 15.15: Codice della funzione `PrintErr` per la generalizzazione della stampa degli errori sullo *standard input* o attraverso il `syslog`.

di errore fornito come argomento sui log di sistema. Se invece si è in modalità interattiva (attivabile con l'opzione `-i`) si usa (5) semplicemente la funzione `perror` per stampare sullo *standard error*.

La gestione del servizio *echo* viene effettuata interamente nella funzione `ServEcho`, il cui codice è mostrato in fig. 15.16, e la comunicazione viene gestita all'interno di un ciclo (6-13). I dati inviati dal client vengono letti (6) dal socket con una semplice `read`, di cui non si controlla lo stato di uscita, assumendo che ritorni solo in presenza di dati in arrivo.

La riscrittura (7) viene invece gestita dalla funzione `FullWrite` (descritta in fig. 15.7) che si incarica di tenere conto automaticamente della possibilità che non tutti i dati di cui è richiesta la scrittura vengano trasmessi con una singola `write`.

In caso di errore di scrittura (si ricordi che `FullWrite` restituisce un valore nullo in caso di

```

1 void ServEcho(int sockfd) {
2     char buffer[MAXLINE];
3     int nread, nwrite;
4     char debug[MAXLINE+20];
5     /* main loop, reading 0 char means client close connection */
6     while ( (nread = read(sockfd, buffer, MAXLINE)) != 0) {
7         nwrite = FullWrite(sockfd, buffer, nread);
8         if (nwrite) {
9             PrintErr("write_error");
10        }
11    }
12    return;
13 }

```

Figura 15.16: Codice della prima versione della funzione ServEcho per la gestione del servizio *echo*.

successo) si provvede (8-10) a stampare il relativo messaggio con `PrintErr`. Quando il client chiude la connessione il ricevimento del segmento FIN fa ritornare la `read` con un numero di byte letti pari a zero, il che causa l'uscita dal ciclo e il ritorno (12) della funzione, che a sua volta causa la terminazione del processo figlio.

15.4.4 L'avvio e il funzionamento ordinario

Benché il codice dell'esempio precedente sia molto ridotto, esso ci permetterà di considerare in dettaglio le varie problematiche che si possono incontrare nello scrivere un'applicazione di rete. Infatti attraverso l'esame delle sue modalità di funzionamento ordinarie, all'avvio e alla terminazione, e di quello che avviene nelle varie situazioni limite, da una parte potremo approfondire la comprensione del protocollo TCP/IP e dall'altra ricavare le indicazioni necessarie per essere in grado di scrivere applicazioni robuste, in grado di gestire anche i casi limite.

Il primo passo è compilare e lanciare il server (usando l'utente di amministrazione, per poter usare la porta 7 che è riservata). All'avvio esso eseguirà l'apertura passiva con la sequenza delle chiamate a `socket`, `bind`, `listen` e poi si bloccherà nella `accept`. A questo punto si potrà controllarne lo stato con `netstat`:

```

[piccardi@roke piccardi]$ netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
...
tcp        0      0 *:echo                  *:                        LISTEN
...

```

che ci mostra come il socket sia in ascolto sulla porta richiesta, accettando connessioni da qualunque indirizzo e da qualunque porta e su qualunque interfaccia locale.

A questo punto si può lanciare il client, esso chiamerà `socket` e `connect`; una volta completato il *three way handshake* la connessione è stabilita; la `connect` ritornerà nel client¹⁶ e la `accept` nel server, ed usando di nuovo `netstat` otterremmo che:

```

Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State

```

¹⁶si noti che è sempre la `connect` del client a ritornare per prima, in quanto questo avviene alla ricezione del secondo segmento (l'ACK del server) del *three way handshake*, la `accept` del server ritorna solo dopo un altro mezzo RTT quando il terzo segmento (l'ACK del client) viene ricevuto.


```

tcp      0      0 *:echo          *: *             LISTEN
tcp      0      0 roke:echo       gont:32981      ESTABLISHED

```

mentre per quanto riguarda l'esecuzione dei programmi avremo che:

- il client chiama la funzione `ClientEcho` che si blocca sulla `fgets` dato che non si è ancora scritto nulla sul terminale.
- il server eseguirà una `fork` facendo chiamare al processo figlio la funzione `ServEcho`, quest'ultima si bloccherà sulla `read` dal socket sul quale ancora non sono presenti dati.
- il processo padre del server chiamerà di nuovo `accept` bloccandosi fino all'arrivo di un'altra connessione.

e se usiamo il comando `ps` per esaminare lo stato dei processi otterremo un risultato del tipo:

```

[piccardi@roke piccardi]$ ps ax
  PID TTY          STAT       TIME COMMAND
  ...  ...          ...        ...   ...
 2356 pts/0    S          0:00    ./echod
 2358 pts/1    S          0:00    ./echo 127.0.0.1
 2359 pts/0    S          0:00    ./echod

```

(dove si sono cancellate le righe inutili) da cui si evidenzia la presenza di tre processi, tutti in stato di *sleep* (vedi tab. 3.8).

Se a questo punto si inizia a scrivere qualcosa sul client non sarà trasmesso niente fintanto che non si preme il tasto di a capo (si ricordi quanto detto in sez. 5.3.5 a proposito dell'I/O su terminale). Solo allora `fgets` ritornerà ed il client scriverà quanto immesso dal terminale sul socket, per poi passare a rileggere quanto gli viene inviato all'indietro dal server, che a sua volta sarà inviato sullo *standard output*, che nel caso ne provoca l'immediata stampa a video.

15.4.5 La conclusione normale

Tutto quello che scriveremo sul client sarà rimandato indietro dal server e ristampato a video fintanto che non concluderemo l'immissione dei dati; una sessione tipica sarà allora del tipo:

```

[piccardi@roke sources]$ ./echo 127.0.0.1
Questa e` una prova
Questa e` una prova
Ho finito
Ho finito

```

che termineremo inviando un EOF dal terminale (usando la combinazione di tasti `ctrl-D`, che non compare a schermo); se eseguiamo un `netstat` a questo punto avremo:

```

[piccardi@roke piccardi]$ netstat -at
tcp      0      0 *:echo          *: *             LISTEN
tcp      0      0 localhost:33032 localhost:echo    TIME_WAIT

```

con il client che entra in `TIME_WAIT`.

Esaminiamo allora in dettaglio la sequenza di eventi che porta alla terminazione normale della connessione, che ci servirà poi da riferimento quando affronteremo il comportamento in caso di conclusioni anomale:

1. inviando un carattere di EOF da terminale la `fgets` ritorna restituendo un puntatore nullo che causa l'uscita dal ciclo di `while`, così la funzione `ClientEcho` ritorna.

2. al ritorno di `ClientEcho` ritorna anche la funzione `main`, e come parte del processo di terminazione tutti i file descriptor vengono chiusi (si ricordi quanto detto in sez. 2.1.5); questo causa la chiusura del socket di comunicazione; il client allora invierà un `FIN` al server a cui questo risponderà con un `ACK`. A questo punto il client verrà a trovarsi nello stato `FIN_WAIT_2` ed il server nello stato `CLOSE_WAIT` (si riveda quanto spiegato in sez. 15.1.3).
3. quando il server riceve il `FIN` la `read` del processo figlio che gestisce la connessione ritorna restituendo 0 causando così l'uscita dal ciclo e il ritorno di `ServEcho`, a questo punto il processo figlio termina chiamando `exit`.
4. all'uscita del figlio tutti i file descriptor vengono chiusi, la chiusura del socket connesso fa sì che venga effettuata la sequenza finale di chiusura della connessione, viene emesso un `FIN` dal server che riceverà un `ACK` dal client, a questo punto la connessione è conclusa e il client resta nello stato `TIME_WAIT`.

15.4.6 La gestione dei processi figli

Tutto questo riguarda la connessione, c'è però da tenere conto dell'effetto del procedimento di chiusura del processo figlio nel server (si veda quanto esaminato in sez. 3.1.4). In questo caso avremo l'invio del segnale `SIGCHLD` al padre, ma dato che non si è installato un gestore e che l'azione predefinita per questo segnale è quella di essere ignorato, non avendo predisposto la ricezione dello stato di terminazione, otterremo che il processo figlio entrerà nello stato di *zombie* (si riveda quanto illustrato in sez. 7.3.6), come risulterà ripetendo il comando `ps`:

```
2356 pts/0    S      0:00 ./echod
2359 pts/0    Z      0:00 [echod <defunct>]
```

Dato che non è il caso di lasciare processi *zombie*, occorrerà ricevere opportunamente lo stato di terminazione del processo (si veda sez. 3.1.5), cosa che faremo utilizzando `SIGCHLD` secondo quanto illustrato in sez. 7.3.6. Una prima modifica al nostro server è pertanto quella di inserire la gestione della terminazione dei processi figli attraverso l'uso di un gestore. Per questo useremo la funzione `Signal` (che abbiamo illustrato in fig. 7.10), per installare il gestore che riceve i segnali dei processi figli terminati già visto in fig. 7.4. Basterà allora aggiungere il seguente codice:

```
...
/* install SIGCHLD handler */
Signal(SIGCHLD, HandSigCHLD); /* establish handler */
/* create socket */
...
```

all'esempio illustrato in fig. 15.13.

In questo modo però si introduce un altro problema. Si ricordi infatti che, come spiegato in sez. 7.3.1, quando un programma si trova in stato di *sleep* durante l'esecuzione di una *system call*, questa viene interrotta alla ricezione di un segnale. Per questo motivo, alla fine dell'esecuzione del gestore del segnale, se questo ritorna, il programma riprenderà l'esecuzione ritornando dalla *system call* interrotta con un errore di `EINTR`.

Vediamo allora cosa comporta tutto questo nel nostro caso: quando si chiude il client, il processo figlio che gestisce la connessione terminerà, ed il padre, per evitare la creazione di *zombie*, riceverà il segnale `SIGCHLD` eseguendo il relativo gestore. Al ritorno del gestore però l'esecuzione nel padre ripartirà subito con il ritorno della funzione `accept` (a meno di un

caso fortuito in cui il segnale arriva durante l'esecuzione del programma in risposta ad una connessione) con un errore di EINTR. Non avendo previsto questa eventualità il programma considera questo un errore fatale terminando a sua volta con un messaggio del tipo:

```
[root@gont sources]# ./echod -i
accept error: Interrupted system call
```

Come accennato in sez. 7.3.1 le conseguenze di questo comportamento delle *system call* possono essere superate in due modi diversi, il più semplice è quello di modificare il codice di `Signal` per richiedere il riavvio automatico delle *system call* interrotte secondo la semantica di BSD, usando l'opzione `SA_RESTART` di `sigaction`; rispetto a quanto visto in fig. 7.10. Definiremo allora la nuova funzione `SignalRestart`¹⁷ come mostrato in fig. 15.17, ed installeremo il gestore usando quest'ultima.

```
1 inline SigFunc * SignalRestart(int signo, SigFunc *func)
2 {
3     struct sigaction new_handler, old_handler;
4     new_handler.sa_handler = func;           /* set signal handler */
5     new_handler.sa_flags = SA_RESTART;      /* restart system call */
6     /* clear signal mask: no signal blocked during execution of func */
7     if (sigemptyset(&new_handler.sa_mask)!=0){ /* initialize signal set */
8         return SIG_ERR;
9     }
10    /* change action for signo signal */
11    if (sigaction(signo, &new_handler, &old_handler)){
12        return SIG_ERR;
13    }
14    return (old_handler.sa_handler);
15 }
```

Figura 15.17: La funzione `SignalRestart`, che installa un gestore di segnali in semantica BSD per il riavvio automatico delle *system call* interrotte.

Come si può notare questa funzione è identica alla precedente `Signal`, illustrata in fig. 7.10, solo che in questo caso invece di inizializzare a zero il campo `sa_flags` di `sigaction`, lo si inizializza (5) al valore `SA_RESTART`. Usando questa funzione al posto di `Signal` nel server non è necessaria nessuna altra modifica: le *system call* interrotte saranno automaticamente riavviate, e l'errore EINTR non si manifesterà più.

La seconda soluzione è più invasiva e richiede di controllare tutte le volte l'errore restituito dalle varie *system call*, ripetendo la chiamata qualora questo corrisponda ad EINTR. Questa soluzione ha però il pregio della portabilità, infatti lo standard POSIX dice che la funzionalità di riavvio automatico delle *system call*, fornita da `SA_RESTART`, è opzionale, per cui non è detto che essa sia disponibile su qualunque sistema. Inoltre in certi casi,¹⁸ anche quando questa è presente, non è detto possa essere usata con `accept`.

La portabilità nella gestione dei segnali però viene al costo di una riscrittura parziale del server, la nuova versione di questo, in cui si sono introdotte una serie di nuove opzioni che ci saranno utili per il debug, è mostrata in fig. 15.18, dove si sono riportate la sezioni di codice modificate nella seconda versione del programma, il codice completo di quest'ultimo si trova nel file `TCP_echod_second.c` dei sorgenti allegati alla guida.

¹⁷anche questa è definita, insieme alle altre funzioni riguardanti la gestione dei segnali, nel file `SigHand.c`, il cui contenuto completo può essere trovato negli esempi allegati.

¹⁸Stevens in [?] accenna che la maggior parte degli Unix derivati da BSD non fanno ripartire `select`; altri non riavviano neanche `accept` e `recvfrom`, cosa che invece nel caso di Linux viene sempre fatta.

La prima modifica effettuata è stata quella di introdurre una nuova opzione a riga di comando, `-c`, che permette di richiedere il comportamento compatibile nella gestione di SIGCHLD al posto della semantica BSD impostando la variabile `compat` ad un valore non nullo. Questa è preimpostata al valore nullo, cosicché se non si usa questa opzione il comportamento di default del server è di usare la semantica BSD.

Una seconda opzione aggiunta è quella di inserire un tempo di attesa fisso specificato in secondi fra il ritorno della funzione `listen` e la chiamata di `accept`, specificabile con l'opzione `-w`, che permette di impostare la variabile `waiting`. Infine si è introdotta una opzione `-d` per abilitare il debugging che imposta ad un valore non nullo la variabile `debugging`. Al solito si è omessa da fig. 15.18 la sezione di codice relativa alla gestione di tutte queste opzioni, che può essere trovata nel sorgente del programma.

Vediamo allora come è cambiato il nostro server; una volta definite le variabili e trattate le opzioni il primo passo (9-13) è verificare la semantica scelta per la gestione di SIGCHLD, a seconda del valore di `compat` (9) si installa il gestore con la funzione `Signal` (10) o con `SignalRestart` (12), essendo quest'ultimo il valore di default.

Tutta la sezione seguente, che crea il socket, cede i privilegi di amministratore ed eventualmente lancia il programma come demone, è rimasta invariata e pertanto è stata omessa in fig. 15.18; l'unica modifica effettuata prima dell'entrata nel ciclo principale è stata quella di aver introdotto, subito dopo la chiamata (17-20) alla funzione `listen`, una eventuale pausa con una condizione (21) sulla variabile `waiting`, che viene inizializzata, con l'opzione `-w Nsec`, al numero di secondi da aspettare (il valore preimpostato è nullo).

Si è potuto lasciare inalterata tutta la sezione di creazione del socket perché nel server l'unica chiamata ad una *system call* lenta, che può essere interrotta dall'arrivo di SIGCHLD, è quella ad `accept`, che è l'unica funzione che può mettere il processo padre in stato di sleep nel periodo in cui un figlio può terminare; si noti infatti come le altre *system call* lente (si ricordi la distinzione fatta in sez. 7.3.1) o sono chiamate prima di entrare nel ciclo principale, quando ancora non esistono processi figli, o sono chiamate dai figli stessi e non risentono di SIGCHLD.

Per questo l'unica modifica sostanziale nel ciclo principale (23-42), rispetto precedente versione di fig. 15.16, è nella sezione (25-31) in cui si effettua la chiamata di `accept`. Quest'ultima viene effettuata (26-27) all'interno di un ciclo di `while`¹⁹ che la ripete indefinitamente qualora in caso di errore il valore di `errno` sia `EINTR`. Negli altri casi si esce in caso di errore effettivo (27-29), altrimenti il programma prosegue.

Si noti che in questa nuova versione si è aggiunta una ulteriore sezione (32-40) di aiuto per il debug del programma, che eseguita con un controllo (33) sul valore della variabile `debugging` impostato dall'opzione `-d`. Qualora questo sia nullo, come preimpostato, non accade nulla, altrimenti (33) l'indirizzo ricevuto da `accept` viene convertito in una stringa che poi (34-39) viene opportunamente stampata o sullo schermo o nei log.

Infine come ulteriore miglione si è perfezionata la funzione `ServEcho`, sia per tenere conto della nuova funzionalità di debugging, che per effettuare un controllo in caso di errore; il codice della nuova versione è mostrato in fig. 15.19.

Rispetto alla precedente versione di fig. 15.16 in questo caso si è provveduto a controllare (7-10) il valore di ritorno di `read` per rilevare un eventuale errore, in modo da stampare (8) un messaggio di errore e ritornare (9) concludendo la connessione.

Inoltre qualora sia stata attivata la funzionalità di debug (avvalorando `debugging` tramite l'apposita opzione `-d`) si provvederà a stampare (16-24) il numero di byte e la stringa letta

¹⁹la sintassi del C relativa a questo ciclo può non essere del tutto chiara. In questo caso infatti si è usato un ciclo vuoto che non esegue nessuna istruzione, in questo modo quello che viene ripetuto con il ciclo è soltanto il codice che esprime la condizione all'interno del `while`.

```

1 int main(int argc, char *argv[])
2 {
3     ...
4     int waiting = 0;
5     int compat = 0;
6     ...
7
8     /* Main code begin here */
9     if (compat) {                                     /* install signal handler */
10         Signal(SIGCHLD, HandSigCHLD);                /* non restarting handler */
11     } else {
12         SignalRestart(SIGCHLD, HandSigCHLD);         /* restarting handler */
13     }
14     ...
15
16     /* main body */
17     if (listen(list_fd, BACKLOG) < 0 ) {
18         PrintErr("listen_error");
19         exit(1);
20     }
21     if (waiting) sleep(waiting);
22     /* handle echo to client */
23     while (1) {
24         /* accept connection */
25         len = sizeof(cli_add);
26         while (((conn_fd = accept(list_fd, (struct sockaddr *)&cli_add, &len))
27                 < 0) && (errno == EINTR));
28         if ( conn_fd < 0) {
29             PrintErr("accept_error");
30             exit(1);
31         }
32         if (debugging) {
33             inet_ntop(AF_INET, &cli_add.sin_addr, ipaddr, sizeof(ipaddr));
34             snprintf(debug, MAXLINE, "Accepted_connection_form_%s\n", ipaddr);
35             if (demonize) {
36                 syslog(LOG_DEBUG, debug);
37             } else {
38                 printf("%s", debug);
39             }
40         }
41         /* fork to handle connection */
42         ...
43     }
44     return;
45 }

```

Figura 15.18: La sezione nel codice della seconda versione del server per il servizio *echo* modificata per tener conto dell'interruzione delle *system call*.

dal client, tenendo conto della modalità di invocazione del server, se interattiva o in forma di demone.

15.5 I vari scenari critici

Con le modifiche viste in sez. 15.4.6 il nostro esempio diventa in grado di affrontare la gestione ordinaria delle connessioni, ma un server di rete deve tenere conto che, al contrario di quanto

```

1 void ServEcho(int sockfd) {
2     char buffer[MAXLINE];
3     int nread, nwrite;
4     char debug[MAXLINE+20];
5     /* main loop, reading 0 char means client close connection */
6     while ( (nread = read(sockfd, buffer, MAXLINE)) != 0) {
7         if (nread < 0) {
8             PrintErr("Errore_in_lettura");
9             return;
10        }
11        nwrite = FullWrite(sockfd, buffer, nread);
12        if (nwrite) {
13            PrintErr("Errore_in_scrittura");
14            return;
15        }
16        if (debugging) {
17            buffer[nread] = 0;
18            snprintf(debug, MAXLINE+20, "Letti_%d_byte,_%s", nread, buffer);
19            if (daemonize) { /* daemon mode */
20                syslog(LOG_DEBUG, debug);
21            } else {
22                printf("%s", debug);
23            }
24        }
25    }
26    return;
27 }

```

Figura 15.19: Codice della seconda versione della funzione ServEcho per la gestione del servizio *echo*.

avviene per i server che operano nei confronti di processi presenti sulla stessa macchina, la rete è di sua natura inaffidabile, per cui è necessario essere in grado di gestire tutta una serie di situazioni critiche che non esistono per i processi locali.

15.5.1 La terminazione precoce della connessione

La prima situazione critica è quella della terminazione precoce, causata da un qualche errore sulla rete, della connessione effettuata da un client. Come accennato in sez. 15.2.4 la funzione `accept` riporta tutti gli eventuali errori di rete pendenti su una connessione sul *connected socket*. Di norma questo non è un problema, in quanto non appena completata la connessione, `accept` ritorna, e l'errore sarà rilevato in seguito dal processo che gestisce la connessione, alla prima chiamata di una funzione che opera sul socket.

È però possibile, dal punto di vista teorico, incorrere anche in uno scenario del tipo di quello mostrato in fig. 15.20, in cui la connessione viene abortita sul lato client per un qualche errore di rete con l'invio di un segmento RST, prima che nel server sia stata chiamata la funzione `accept`.

Figura 15.20: Un possibile caso di terminazione precoce della connessione.

Benché questo non sia un fatto comune, un evento simile può essere osservato con dei server molto occupati. In tal caso, con una struttura del server simile a quella del nostro esempio, in cui la gestione delle singole connessioni è demandata a processi figli, può accadere

che il *three way handshake* venga completato e la relativa connessione abortita subito dopo, prima che il padre, per via del carico della macchina, abbia fatto in tempo ad eseguire la chiamata ad `accept`. Di nuovo si ha una situazione analoga a quella illustrata in fig. 15.20, in cui la connessione viene stabilita, ma subito dopo si ha una condizione di errore che la chiude prima che essa sia stata accettata dal programma.

Questo significa che oltre alla interruzione da parte di un segnale, che abbiamo trattato in sez. 15.4.6 nel caso particolare di `SIGCHLD`, si possono ricevere altri errori non fatali all'uscita di `accept`; questi, come nel caso precedente, necessitano semplicemente la ripetizione della chiamata senza uscire dal programma. In questo caso anche la versione modificata del nostro server non sarebbe adatta, in quanto uno di questi errori causerebbe la terminazione dello stesso. In Linux i possibili errori di rete non fatali, riportati sul socket connesso al ritorno di `accept`, sono `ENETDOWN`, `EPROTO`, `ENOPROTOOPT`, `EHOSTDOWN`, `ENONET`, `EHOSTUNREACH`, `EOPNOTSUPP` e `ENETUNREACH`.

Si tenga presente che questo tipo di terminazione non è riproducibile terminando il client prima della chiamata ad `accept`, come si potrebbe fare usando l'opzione `-w` per introdurre una pausa dopo il lancio del demone, in modo da poter avere il tempo per lanciare e terminare una connessione usando il programma client. In tal caso infatti, alla terminazione del client, il socket associato alla connessione viene semplicemente chiuso, attraverso la sequenza vista in sez. 15.1.3, per cui la `accept` ritornerà senza errori, e si avrà semplicemente un *end-of-file* al primo accesso al socket. Nel caso di Linux inoltre, anche qualora si modifichi il client per fargli gestire l'invio di un segmento `RST` alla chiusura dal socket (usando l'opzione `SO_LINGER`, vedi sez. 16.2.3), non si ha comunque nessun errore al ritorno di `accept`, quanto un errore di `ECONNRESET` al primo tentativo di accesso al socket.

15.5.2 La terminazione precoce del server

Un secondo caso critico è quello in cui si ha una terminazione precoce del server, ad esempio perché il programma ha un crash. In tal caso si suppone che il processo termini per un errore fatale, cosa che potremo simulare inviandogli un segnale di terminazione. La conclusione del processo comporta la chiusura di tutti i file descriptor aperti, compresi tutti i socket relativi a connessioni stabilite; questo significa che al momento del crollo del servizio il client riceverà un `FIN` dal server in corrispondenza della chiusura del socket.

Vediamo allora cosa succede nel nostro caso, facciamo partire una connessione con il server e scriviamo una prima riga, poi terminiamo il server con un `C-c`. A questo punto scriviamo una seconda riga e poi un'altra riga ancora. Il risultato finale della sessione è il seguente:

```
[piccardi@gont sources]$ ./echo 192.168.1.141
Prima riga
Prima riga
Seconda riga dopo il C-c
Altra riga
[piccardi@gont sources]$
```

Come si vede il nostro client, nonostante la connessione sia stata interrotta prima dell'invio della seconda riga, non solo accetta di inviarla, ma prende anche un'altra riga prima di terminare senza riportare nessun errore.

Per capire meglio cosa è successo conviene analizzare il flusso dei pacchetti utilizzando un analizzatore di traffico come `tcpdump`. Il comando permette di selezionare, nel traffico di rete generato su una macchina, i pacchetti che interessano, stampando a video (o salvando su disco) il loro contenuto. Non staremo qui ad entrare nei dettagli dell'uso del programma, che

sono spiegati dalla pagina di manuale;²⁰ per l'uso che vogliamo farne quello che ci interessa è, posizionandosi sulla macchina che fa da client, selezionare tutti i pacchetti che sono diretti o provengono dalla macchina che fa da server. In questo modo (posto che non ci siano altre connessioni col server, cosa che avremo cura di evitare) tutti i pacchetti rilevati apparterranno alla nostra sessione di interrogazione del servizio.

Il comando `tcpdump` permette selezioni molto complesse, basate sulle interfacce su cui passano i pacchetti, sugli indirizzi IP, sulle porte, sulle caratteristiche ed il contenuto dei pacchetti stessi, inoltre permette di combinare fra loro diversi criteri di selezione con degli operatori logici; quando un pacchetto che corrisponde ai criteri di selezione scelti viene rilevato i suoi dati vengono stampati sullo schermo (anche questi secondo un formato configurabile in maniera molto precisa).

Lanciando il comando prima di ripetere la sessione di lavoro mostrata nell'esempio precedente potremo allora catturare tutti pacchetti scambiati fra il client ed il server; i risultati (in realtà si è ridotta la lunghezza dell'output rispetto al reale tagliando alcuni dati non necessari alla comprensione del flusso) prodotti in questa occasione da `tcpdump` sono allora i seguenti:

```
[root@gont gapil]# tcpdump src 192.168.1.141 or dst 192.168.1.141 -N -t
tcpdump: listening on eth0
gont.34559 > anarres.echo: S 800922320:800922320(0) win 5840
anarres.echo > gont.34559: S 511689719:511689719(0) ack 800922321 win 5792
gont.34559 > anarres.echo: . ack 1 win 5840
gont.34559 > anarres.echo: P 1:12(11) ack 1 win 5840
anarres.echo > gont.34559: . ack 12 win 5792
anarres.echo > gont.34559: P 1:12(11) ack 12 win 5792
gont.34559 > anarres.echo: . ack 12 win 5840
anarres.echo > gont.34559: F 12:12(0) ack 12 win 5792
gont.34559 > anarres.echo: . ack 13 win 5840
gont.34559 > anarres.echo: P 12:37(25) ack 13 win 5840
anarres.echo > gont.34559: R 511689732:511689732(0) win 0
```

Le prime tre righe vengono prodotte al momento in cui lanciamo il nostro client, e corrispondono ai tre pacchetti del *three way handshake*. L'output del comando riporta anche i numeri di sequenza iniziali, mentre la lettera S indica che per quel pacchetto si aveva il SYN flag attivo. Si noti come a partire dal secondo pacchetto sia sempre attivo il campo ack, seguito dal numero di sequenza per il quale si dà il ricevuto; quest'ultimo, a partire dal terzo pacchetto, viene espresso in forma relativa per maggiore compattezza. Il campo win in ogni riga indica la *advertised window* di cui parlavamo in sez. 15.1.2.

Allora si può verificare dall'output del comando come venga appunto realizzata la sequenza di pacchetti descritta in sez. 15.1.1: prima viene inviato dal client un primo pacchetto con il SYN che inizia la connessione, a cui il server risponde dando il ricevuto con un secondo pacchetto, che a sua volta porta un SYN, cui il client risponde con un il terzo pacchetto di ricevuto.

Ritorniamo allora alla nostra sessione con il servizio echo: dopo le tre righe del *three way handshake* non avremo nulla fin tanto che non scriveremo una prima riga sul client; al momento in cui facciamo questo si genera una sequenza di altri quattro pacchetti. Il primo, dal client al server, contraddistinto da una lettera P che significa che il flag PSH è impostato, contiene la nostra riga (che è appunto di 11 caratteri), e ad esso il server risponde immediatamente con un pacchetto vuoto di ricevuto.

Poi tocca al server riscrivere indietro quanto gli è stato inviato, per cui sarà lui a mandare indietro un terzo pacchetto con lo stesso contenuto appena ricevuto, e a sua volta riceverà dal client un ACK nel quarto pacchetto. Questo causerà la ricezione dell'eco nel client che lo stamperà a video.

²⁰per una trattazione di base si può consultare sez. 5.2.2 di [?].

A questo punto noi procediamo ad interrompere l'esecuzione del server con un C-c (cioè con l'invio di SIGTERM): nel momento in cui facciamo questo vengono immediatamente generati altri due pacchetti. La terminazione del processo infatti comporta la chiusura di tutti i suoi file descriptor, il che comporta, per il socket che avevamo aperto, l'inizio della sequenza di chiusura illustrata in sez. 15.1.3. Questo significa che dal server partirà un FIN, che è appunto il primo dei due pacchetti, contraddistinto dalla lettera F, cui seguirà al solito un ACK da parte del client.

A questo punto la connessione dalla parte del server è chiusa, ed infatti se usiamo `netstat` per controllarne lo stato otterremo che sul server si ha:

```
anarres:/home/piccardi# netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
...      ...      ...
tcp        0      0 192.168.1.141:7        192.168.1.2:34626      FIN_WAIT2
```

cioè essa è andata nello stato `FIN_WAIT2`, che indica l'avvenuta emissione del segmento FIN, mentre sul client otterremo che essa è andata nello stato `CLOSE_WAIT`:

```
[root@gont gapil]# netstat -ant
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
...      ...      ...
tcp        1      0 192.168.1.2:34582      192.168.1.141:7        CLOSE_WAIT
```

Il problema è che in questo momento il client è bloccato dentro la funzione `ClientEcho` nella chiamata a `fgets`, e sta attendendo dell'input dal terminale, per cui non è in grado di accorgersi di nulla. Solo quando inseriremo la seconda riga il comando uscirà da `fgets` e proverà a scriverla sul socket. Questo comporta la generazione degli ultimi due pacchetti riportati da `tcpdump`: il primo, inviato dal client contenente i 25 caratteri della riga appena letta, e ad esso la macchina server risponderà, non essendoci più niente in ascolto sulla porta 7, con un segmento di RST, contraddistinto dalla lettera R, che causa la conclusione definitiva della connessione anche nel client, dove non comparirà più nell'output di `netstat`.

Come abbiamo accennato in sez. 15.1.3 e come vedremo più avanti in sez. 15.6.3 la chiusura di un solo capo di un socket è una operazione lecita, per cui la nostra scrittura avrà comunque successo (come si può constatare lanciando usando `strace`²¹), in quanto il nostro programma non ha a questo punto alcun modo di sapere che dall'altra parte non c'è più nessuno processo in grado di leggere quanto scriverà. Questo sarà chiaro solo dopo il tentativo di scrittura, e la ricezione del segmento RST di risposta che indica che dall'altra parte non si è semplicemente chiuso un capo del socket, ma è completamente terminato il programma.

Per questo motivo il nostro client proseguirà leggendo dal socket, e dato che questo è stato chiuso avremo che, come spiegato in sez. 15.1.3, la funzione `read` ritorna normalmente con un valore nullo. Questo comporta che la seguente chiamata a `fputs` non ha effetto (viene stampata una stringa nulla) ed il client si blocca di nuovo nella successiva chiamata a `fgets`. Per questo diventa possibile inserire una terza riga e solo dopo averlo fatto si avrà la terminazione del programma.

Per capire come questa avvenga comunque, non avendo inserito nel codice nessun controllo di errore, occorre ricordare che, a parte la bidirezionalità del flusso dei dati, dal punto di vista del funzionamento nei confronti delle funzioni di lettura e scrittura, i socket sono del tutto

²¹il comando `strace` è un comando di debug molto utile che prende come argomento un altro comando e ne stampa a video tutte le invocazioni di una *system call*, coi relativi argomenti e valori di ritorno, per cui usandolo in questo contesto potremo verificare che effettivamente la `write` ha scritto la riga, che in effetti è stata pure trasmessa via rete.

analoghi a delle *pipe*. Allora, da quanto illustrato in sez. 11.1.1, sappiamo che tutte le volte che si cerca di scrivere su una *pipe* il cui altro capo non è aperto il lettura il processo riceve un segnale di SIGPIPE, e questo è esattamente quello che avviene in questo caso, e siccome non abbiamo un gestore per questo segnale, viene eseguita l'azione preimpostata, che è quella di terminare il processo.

Per gestire in maniera più corretta questo tipo di evento dovremo allora modificare il nostro client perché sia in grado di trattare le varie tipologie di errore, per questo dovremo riscrivere la funzione ClientEcho, in modo da controllare gli stati di uscita delle varie chiamate. Si è riportata la nuova versione della funzione in fig. 15.21.

```

1 void ClientEcho(FILE * filein, int socket)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread, nwrite;
5     while (fgets(sendbuff, MAXLINE, filein) != NULL) {
6         nwrite = FullWrite(socket, sendbuff, strlen(sendbuff));
7         if (nwrite < 0) {
8             printf("Errore_in_scrittura:_%s", strerror(errno));
9             return;
10        }
11        nread = read(socket, recvbuff, strlen(sendbuff));
12        if (nread < 0) {
13            printf("Errore_in_lettura:_%s\n", strerror(errno));
14            return;
15        }
16        if (nread == 0) {
17            printf("End_of_file_in_lettura_%s\n");
18            return;
19        }
20        recvbuff[nread] = 0;
21        if (fputs(recvbuff, stdout) == EOF) {
22            perror("Errore_in_scrittura_su_terminale");
23            return;
24        }
25    }
26    return;
27 }

```

Figura 15.21: La sezione nel codice della seconda versione della funzione ClientEcho usata dal client per il servizio *echo* modificata per tener conto degli eventuali errori.

Come si può vedere in questo caso si controlla il valore di ritorno di tutte le funzioni, ed inoltre si verifica la presenza di un eventuale *end-of-file* in caso di lettura. Con questa modifica il nostro client echo diventa in grado di accorgersi della chiusura del socket da parte del server, per cui ripetendo la sequenza di operazioni precedenti stavolta otterremo che:

```

[piccardi@gont sources]$ ./echo 192.168.1.141
Prima riga
Prima riga
Seconda riga dopo il C-c
EOF sul socket

```

ma di nuovo si tenga presente che non c'è modo di accorgersi della chiusura del socket fin quando non si esegue la scrittura della seconda riga; il protocollo infatti prevede che ci debba essere una scrittura prima di ricevere un RST che confermi la chiusura del file, e solo alle successive scritture si potrà ottenere un errore.

Questa caratteristica dei socket ci mette di fronte ad un altro problema relativo al nostro client, e che cioè esso non è in grado di accorgersi di nulla fintanto che è bloccato nella lettura del terminale fatta con `gets`. In questo caso il problema è minimo, ma esso riemergerà più avanti, ed è quello che si deve affrontare tutte le volte quando si ha a che fare con la necessità di lavorare con più descrittori, nel qual caso si pone la questione di come fare a non restare bloccati su un socket quando altri potrebbero essere liberi. Vedremo come affrontare questa problematica in sez. 15.6.

15.5.3 Altri scenari di terminazione della connessione

La terminazione del server è solo uno dei possibili scenari di terminazione della connessione, un altro caso è ad esempio quello in cui si ha un'interruzione sulla rete, cosa che potremo simulare facilmente staccando il cavo di rete. Un'altra condizione è quella di un blocco completo della macchina su cui gira il server che deve essere riavviata, cosa che potremo simulare sia eseguendo un reset fisico (un normale shutdown non va bene; in tal caso infatti il sistema provvede a terminare tutti i processi, per cui la situazione sarebbe sostanzialmente identica alla precedente) oppure, in maniera più gentile, riavviando la macchina dopo aver interrotto la connessione di rete.

Cominciamo ad analizzare il primo caso, l'interruzione del collegamento di rete. Ripetiamo la nostra sessione di lavoro precedente, lanciamo il client, scriviamo una prima riga, poi stacciamo il cavo e scriviamo una seconda riga. Il risultato che otterremo è:

```
[piccardi@gont sources]$ ./echo 192.168.1.141
Prima riga
Prima riga
Seconda riga dopo l'interruzione
Errore in lettura: No route to host
```

Quello che succede in questo è che il programma, dopo aver scritto la seconda riga, resta bloccato per un tempo molto lungo, prima di dare l'errore `EHOSTUNREACH`. Se andiamo ad osservare con `strace` cosa accade nel periodo in cui il programma è bloccato vedremo che stavolta, a differenza del caso precedente, il programma è bloccato nella lettura dal socket.

Se poi, come nel caso precedente, usiamo l'accortezza di analizzare il traffico di rete fra client e server con `tcpdump`, otterremo il seguente risultato:

```
[root@gont sources]# tcpdump src 192.168.1.141 or dst 192.168.1.141 -N -t
tcpdump: listening on eth0
gont.34685 > anarres.echo: S 1943495663:1943495663(0) win 5840
anarres.echo > gont.34685: S 1215783131:1215783131(0) ack 1943495664 win 5792
gont.34685 > anarres.echo: . ack 1 win 5840
gont.34685 > anarres.echo: P 1:12(11) ack 1 win 5840
anarres.echo > gont.34685: . ack 12 win 5792
anarres.echo > gont.34685: P 1:12(11) ack 12 win 5792
gont.34685 > anarres.echo: . ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
arp who-has anarres tell gont
arp who-has anarres tell gont
arp who-has anarres tell gont
```

```
arp who-has anarres tell gont
arp who-has anarres tell gont
arp who-has anarres tell gont
...
```

In questo caso l'andamento dei primi sette pacchetti è esattamente lo stesso di prima. Solo che stavolta, non appena inviata la seconda riga, il programma si bloccherà nella successiva chiamata a `read`, non ottenendo nessuna risposta. Quello che succede è che nel frattempo il kernel provvede, come richiesto dal protocollo TCP, a tentare la ritrasmissione della nostra riga un certo numero di volte, con tempi di attesa crescente fra un tentativo ed il successivo, per tentare di ristabilire la connessione.

Il risultato finale qui dipende dalla realizzazione dello *stack TCP*, e nel caso di Linux anche dall'impostazione di alcuni dei parametri di sistema che si trovano in `/proc/sys/net/ipv4`, che ne controllano il comportamento: in questo caso in particolare da `tcp_retries2` (vedi sez. 16.4.3). Questo parametro infatti specifica il numero di volte che deve essere ritentata la ritrasmissione di un pacchetto nel mezzo di una connessione prima di riportare un errore di timeout. Il valore preimpostato è pari a 15, il che comporterebbe 15 tentativi di ritrasmissione, ma nel nostro caso le cose sono andate diversamente, dato che le ritrasmissioni registrate da `tcpdump` sono solo 8; inoltre l'errore riportato all'uscita del client non è stato `ETIMEDOUT`, come dovrebbe essere in questo caso, ma `EHOSTUNREACH`.

Per capire l'accaduto continuiamo ad analizzare l'output di `tcpdump`: esso ci mostra che a un certo punto i tentativi di ritrasmissione del pacchetto sono cessati, per essere sostituiti da una serie di richieste di protocollo ARP in cui il client richiede l'indirizzo del server.

Come abbiamo accennato in sez. 13.3.1 ARP è il protocollo che si incarica di trovare le corrispondenze fra indirizzo IP e indirizzo hardware sulla scheda di rete. È evidente allora che nel nostro caso, essendo client e server sulla stessa rete, è scaduta la voce nella *ARP cache*²² relativa ad `anarres`, ed il nostro client ha iniziato ad effettuare richieste ARP sulla rete per sapere l'IP di quest'ultimo, che essendo scollegato non poteva rispondere. Anche per questo tipo di richieste esiste un timeout, per cui dopo un certo numero di tentativi il meccanismo si è interrotto, e l'errore riportato al programma a questo punto è stato `EHOSTUNREACH`, in quanto non si era più in grado di contattare il server.

Un altro errore possibile in questo tipo di situazione, che si può avere quando la macchina è su una rete remota, è `ENETUNREACH`; esso viene riportato alla ricezione di un pacchetto ICMP di *destination unreachable* da parte del router che individua l'interruzione della connessione. Di nuovo anche qui il risultato finale dipende da quale è il meccanismo più veloce che porta ad accorgersi del problema.

Se però agiamo sui parametri del kernel, e scriviamo in `tcp_retries2` un valore di tentativi più basso, possiamo evitare la scadenza della *ARP cache* e vedere cosa succede. Così se ad esempio richiediamo 4 tentativi di ritrasmissione, l'analisi di `tcpdump` ci riporterà il seguente scambio di pacchetti:

```
[root@gont gapil]# tcpdump src 192.168.1.141 or dst 192.168.1.141 -N -t
tcpdump: listening on eth0
gont.34752 > anarres.echo: S 3646972152:3646972152(0) win 5840
anarres.echo > gont.34752: S 2735190336:2735190336(0) ack 3646972153 win 5792
gont.34752 > anarres.echo: . ack 1 win 5840
gont.34752 > anarres.echo: P 1:12(11) ack 1 win 5840
anarres.echo > gont.34752: . ack 12 win 5792
anarres.echo > gont.34752: P 1:12(11) ack 12 win 5792
```

²²la *ARP cache* è una tabella mantenuta internamente dal kernel che contiene tutte le corrispondenze fra indirizzi IP e indirizzi fisici, ottenute appunto attraverso il protocollo ARP; le voci della tabella hanno un tempo di vita limitato, passato il quale scadono e devono essere nuovamente richieste.

```

gont.34752 > anarres.echo: . ack 12 win 5840
gont.34752 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34752 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34752 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34752 > anarres.echo: P 12:45(33) ack 12 win 5840
gont.34752 > anarres.echo: P 12:45(33) ack 12 win 5840

```

e come si vede in questo caso i tentativi di ritrasmissione del pacchetto iniziale sono proprio 4 (per un totale di 5 voci con quello trasmesso la prima volta), ed in effetti, dopo un tempo molto più breve rispetto a prima ed in corrispondenza dell'invio dell'ultimo tentativo, quello che otterremo come errore all'uscita del client sarà diverso, e cioè:

```

[piccardi@gont sources]$ ./echo 192.168.1.141
Prima riga
Prima riga
Seconda riga dopo l'interruzione
Errore in lettura: Connection timed out

```

che corrisponde appunto, come ci aspettavamo, alla ricezione di un ETIMEDOUT.

Analizziamo ora il secondo scenario, in cui si ha un crollo della macchina che fa da server. Al solito lanciamo il nostro client, scriviamo una prima riga per verificare che sia tutto a posto, poi stacciamo il cavo e riavviamo il server. A questo punto, ritornato attivo il server, scriviamo una seconda riga. Quello che otterremo in questo caso è:

```

[piccardi@gont sources]$ ./echo 192.168.1.141
Prima riga
Prima riga
Seconda riga dopo l'interruzione
Errore in lettura Connection reset by peer

```

e l'errore ricevuto da read stavolta è ECONNRESET. Se al solito riportiamo l'analisi dei pacchetti effettuata con `tcpdump`, avremo:

```

[root@gont gapil]# tcpdump src 192.168.1.141 or dst 192.168.1.141 -N -t
tcpdump: listening on eth0
gont.34756 > anarres.echo: S 904864257:904864257(0) win 5840
anarres.echo > gont.34756: S 4254564871:4254564871(0) ack 904864258 win 5792
gont.34756 > anarres.echo: . ack 1 win 5840
gont.34756 > anarres.echo: P 1:12(11) ack 1 win 5840
anarres.echo > gont.34756: . ack 12 win 5792
anarres.echo > gont.34756: P 1:12(11) ack 12 win 5792
gont.34756 > anarres.echo: . ack 12 win 5840
gont.34756 > anarres.echo: P 12:45(33) ack 12 win 5840
anarres.echo > gont.34756: R 4254564883:4254564883(0) win 0

```

Ancora una volta i primi sette pacchetti sono gli stessi; ma in questo caso quello che succede dopo lo scambio iniziale è che, non avendo inviato nulla durante il periodo in cui si è riavviato il server, il client è del tutto ignaro dell'accaduto per cui quando effettuerà una scrittura, dato che la macchina server è stata riavviata e che tutti gli stati relativi alle precedenti connessioni sono completamente persi, anche in presenza di una nuova istanza del server echo non sarà possibile consegnare i dati in arrivo, per cui alla loro ricezione il kernel risponderà con un segmento di RST.

Il client da parte sua, dato che neanche in questo caso non è stato emesso un FIN, dopo aver scritto verrà bloccato nella successiva chiamata a `read`, che però adesso ritornerà immediatamente alla ricezione del segmento RST, riportando appunto come errore ECONNRESET. Occorre precisare che se si vuole che il client sia in grado di accorgersi del crollo del server anche quando non sta effettuando uno scambio di dati, è possibile usare una impostazione speciale del socket (ci torneremo in sez. 16.2.2) che provvede all'esecuzione di questo controllo.

15.6 L'uso dell'I/O *multiplexing*

Affronteremo in questa sezione l'utilizzo dell'I/O *multiplexing*, affrontato in sez. 10.2, nell'ambito delle applicazioni di rete. Già in sez. 15.5.2 era emerso il problema relativo al client del servizio *echo* che non era in grado di accorgersi della terminazione precoce del server, essendo bloccato nella lettura dei dati immessi da tastiera.

Abbiamo visto in sez. 10.2 quali sono le funzionalità del sistema che ci permettono di tenere sotto controllo più file descriptor in contemporanea; in quella occasione non abbiamo fatto esempi, in quanto quando si tratta con file normali questa tipologia di I/O normalmente non viene usata, è invece un caso tipico delle applicazioni di rete quello di dover gestire varie connessioni da cui possono arrivare dati comuni in maniera asincrona, per cui riprenderemo l'argomento in questa sezione.

15.6.1 Il comportamento della funzione `select` con i socket.

Iniziamo con la prima delle funzioni usate per l'I/O *multiplexing*, `select`; il suo funzionamento è già stato descritto in dettaglio in sez. 10.2 e non staremo a ripetere quanto detto lì; sappiamo che la funzione ritorna quando uno o più dei file descriptor messi sotto controllo è pronto per la relativa operazione.

In quell'occasione non abbiamo però definito cosa si intende per pronto, infatti per dei normali file, o anche per delle *pipe*, la condizione di essere pronti per la lettura o la scrittura è ovvia; invece lo è molto meno nel caso dei socket, visto che possono intervenire tutte una serie di possibili condizioni di errore dovute alla rete. Occorre allora specificare chiaramente quali sono le condizioni per cui un socket risulta essere “pronto” quando viene passato come membro di uno dei tre *file descriptor set* usati da `select`.

Le condizioni che fanno sì che la funzione `select` ritorni segnalando che un socket (che sarà riportato nel primo insieme di file descriptor) è pronto per la lettura sono le seguenti:

- nel buffer di ricezione del socket sono arrivati dei dati in quantità sufficiente a superare il valore di una *soglia di basso livello* (il cosiddetto *low watermark*). Questo valore è espresso in numero di byte e può essere impostato con l'opzione del socket `SO_RCVLOWAT` (tratteremo l'uso di questa opzione in sez. 16.2.2); il suo valore di default è 1 per i socket TCP e UDP. In questo caso una operazione di lettura avrà successo e leggerà un numero di byte maggiore di zero.
- il lato in lettura della connessione è stato chiuso; si è cioè ricevuto un segmento FIN (si ricordi quanto illustrato in sez. 15.1.3) sulla connessione. In questo caso una operazione di lettura avrà successo, ma non risulteranno presenti dati (in sostanza `read` ritornerà con un valore nullo) per indicare la condizione di *end-of-file*.
- c'è stato un errore sul socket. In questo caso una operazione di lettura non si bloccherà ma restituirà una condizione di errore (ad esempio `read` restituirà `-1`) e imposterà la variabile `errno` al relativo valore. Vedremo in sez. 16.2.2 come sia possibile estrarre e cancellare gli errori pendenti su un socket senza usare `read` usando l'opzione `SO_ERROR`.
- quando si sta utilizzando un *listening socket* ed ci sono delle connessioni completate. In questo caso la funzione `accept` non si bloccherà.²³

Le condizioni che fanno sì che la funzione `select` ritorni segnalando che un socket (che sarà riportato nel secondo insieme di file descriptor) è pronto per la scrittura sono le seguenti:

²³in realtà questo non è sempre vero, come accennato in sez. 15.5.1 una connessione può essere abortita dalla ricezione di un segmento RST una volta che è stata completata, allora se questo avviene dopo che `select` è ritornata, ma prima della chiamata ad `accept`, quest'ultima, in assenza di altre connessioni, potrà bloccarsi.

- nel buffer di invio è disponibile una quantità di spazio superiore al valore della *soglia di basso livello* in scrittura ed inoltre o il socket è già connesso o non necessita (ad esempio è UDP) di connessione. Il valore della soglia è espresso in numero di byte e può essere impostato con l'opzione del socket `SO_SNDLOWAT` (trattata in sez. 16.2.2); il suo valore di default è 2048 per i socket TCP e UDP. In questo caso una operazione di scrittura non si bloccherà e restituirà un valore positivo pari al numero di byte accettati dal livello di trasporto.
- il lato in scrittura della connessione è stato chiuso. In questo caso una operazione di scrittura sul socket genererà il segnale `SIGPIPE`.
- c'è stato un errore sul socket. In questo caso una operazione di scrittura non si bloccherà ma restituirà una condizione di errore ed imposterà opportunamente la variabile `errno`. Vedremo in sez. 16.2.2 come sia possibile estrarre e cancellare errori pendenti su un socket usando l'opzione `SO_ERROR`.

Infine c'è una sola condizione che fa sì che `select` ritorni segnalando che un socket (che sarà riportato nel terzo insieme di file descriptor) ha una condizione di eccezione pendente, e cioè la ricezione sul socket di *dati urgenti* (o *out-of-band*), una caratteristica specifica dei socket TCP su cui torneremo in sez. 18.1.4.

Si noti come nel caso della lettura `select` si applichi anche ad operazioni che non hanno nulla a che fare con l'I/O di dati come il riconoscimento della presenza di connessioni pronte, in modo da consentire anche l'utilizzo di `accept` in modalità non bloccante. Si noti infine come in caso di errore un socket venga sempre riportato come pronto sia per la lettura che per la scrittura.

Lo scopo dei due valori di soglia per i buffer di ricezione e di invio è quello di consentire maggiore flessibilità nell'uso di `select` da parte dei programmi, se infatti si sa che una applicazione non è in grado di fare niente fintanto che non può ricevere o inviare una certa quantità di dati, si possono utilizzare questi valori per far sì che `select` ritorni solo quando c'è la certezza di avere dati a sufficienza.²⁴

15.6.2 Un esempio di *I/O multiplexing*

Abbiamo incontrato la problematica tipica che conduce all'uso dell'*I/O multiplexing* nella nostra analisi degli errori in sez. 15.5.1, quando il nostro client non era in grado di rendersi conto di errori sulla connessione essendo impegnato nella attesa di dati in ingresso dallo *standard input*.

In questo caso il problema è quello di dover tenere sotto controllo due diversi file descriptor, lo *standard input*, da cui viene letto il testo che vogliamo inviare al server, e il socket connesso con il server su cui detto testo sarà scritto e dal quale poi si vorrà ricevere la risposta. L'uso dell'*I/O multiplexing* consente di tenere sotto controllo entrambi, senza restare bloccati.

Nel nostro caso quello che ci interessa è non essere bloccati in lettura sullo *standard input* in caso di errori sulla connessione o chiusura della stessa da parte del server. Entrambi questi casi possono essere rilevati usando `select`, per quanto detto in sez. 15.6.1, mettendo sotto osservazione i file descriptor per la condizione di essere pronti in lettura: sia infatti che si ricevano dati, che la connessione sia chiusa regolarmente (con la ricezione di un segmento

²⁴questo tipo di controllo è utile di norma solo per la lettura, in quanto in genere le operazioni di scrittura sono già controllate dall'applicazione, che sa sempre quanti dati invia, mentre non è detto possa conoscere la quantità di dati in ricezione; per cui, nella situazione in cui si conosce almeno un valore minimo, per evitare la penalizzazione dovuta alla ripetizione delle operazioni di lettura per accumulare dati sufficienti, si può lasciare al kernel il compito di impostare un minimo al di sotto del quale il socket, pur avendo disponibili dei dati, non viene dato per pronto in lettura.

FIN) che si riceva una condizione di errore (con un segmento RST) il socket connesso sarà pronto in lettura (nell'ultimo caso anche in scrittura, ma questo non è necessario ai nostri scopi).

```

1 void ClientEcho(FILE * filein, int socket)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread, nwrite;
5     int maxfd;
6     fd_set fset;
7     /* initialize file descriptor set */
8     FD_ZERO(&fset);
9     maxfd = max(fileno(filein), socket) + 1;
10    while (1) {
11        FD_SET(socket, &fset); /* set for the socket */
12        FD_SET(fileno(filein), &fset); /* set for the standard input */
13        select(maxfd, &fset, NULL, NULL, NULL); /* wait for read ready */
14        if (FD_ISSET(fileno(filein), &fset)) { /* if ready on stdin */
15            if (fgets(sendbuff, MAXLINE, filein) == NULL) { /* if no input */
16                return; /* we stopped client */
17            } else { /* else we have to write to socket */
18                nwrite = FullWrite(socket, sendbuff, strlen(sendbuff));
19                if (nwrite < 0) { /* on error stop */
20                    printf("Errore_in_scrittura:_%s", strerror(errno));
21                    return;
22                }
23            }
24        }
25        if (FD_ISSET(socket, &fset)) { /* if ready on socket */
26            nread = read(socket, recvbuff, strlen(sendbuff)); /* do read */
27            if (nread < 0) { /* error condition, stop client */
28                printf("Errore_in_lettura:_%s\n", strerror(errno));
29                return;
30            }
31            if (nread == 0) { /* server closed connection, stop */
32                printf("EOF_sul_socket\n");
33                return;
34            }
35            recvbuff[nread] = 0; /* else read is ok, write on stdout */
36            if (fputs(recvbuff, stdout) == EOF) {
37                perror("Errore_in_scrittura_su_terminale");
38                return;
39            }
40        }
41    }
42 }

```

Figura 15.22: La sezione nel codice della terza versione della funzione ClientEcho usata dal client per il servizio *echo* modificata per l'uso di *select*.

Riprendiamo allora il codice del client, modificandolo per l'uso di *select*. Quello che dobbiamo modificare è la funzione ClientEcho di fig. 15.21, dato che tutto il resto, che riguarda le modalità in cui viene stabilita la connessione con il server, resta assolutamente identico. La nostra nuova versione di ClientEcho, la terza della serie, è riportata in fig. 15.22, il codice completo si trova nel file TCP_echo_third.c dei sorgenti allegati alla guida.

In questo caso la funzione comincia (8-9) con l'azzeramento del *file descriptor set* *fset*

e l'impostazione del valore `maxfd`, da passare a `select` come massimo per il numero di file descriptor. Per determinare quest'ultimo si usa la macro `max` definita nel nostro file `macro.h` che raccoglie una collezione di macro di preprocessore di varia utilità.

La funzione prosegue poi (10-41) con il ciclo principale, che viene ripetuto indefinitamente. Per ogni ciclo si reinizializza (11-12) il *file descriptor set*, impostando i valori per il file descriptor associato al socket `socket` e per lo *standard input* (il cui valore si recupera con la funzione `fileno`). Questo è necessario in quanto la successiva (13) chiamata a `select` comporta una modifica dei due bit relativa questi file, che quindi devono essere reimpostati all'inizio di ogni ciclo.

Si noti come la chiamata a `select` venga eseguita usando come primo argomento il valore di `maxfd`, precedentemente calcolato, e passando poi il solo *file descriptor set* per il controllo dell'attività in lettura, negli altri argomenti vengono passati tutti puntatori nulli, non interessando in questo caso né il controllo delle altre attività, né l'impostazione di un valore di timeout.

Al ritorno di `select` si provvede a controllare quale dei due file descriptor presenta attività in lettura, cominciando (14-24) con il file descriptor associato allo *standard input*. In caso di attività (quando cioè `FD_ISSET` ritorna un valore diverso da zero) si esegue (15) una `fgets` per leggere gli eventuali dati presenti; se non ve ne sono (e la funzione restituisce pertanto un puntatore nullo) si ritorna immediatamente (16) dato che questo significa che si è chiuso lo *standard input* e quindi concluso l'utilizzo del client; altrimenti (18-22) si scrivono i dati appena letti sul socket, prevedendo una uscita immediata in caso di errore di scrittura.

Controllato lo *standard input* si passa a controllare (25-40) il socket connesso, in caso di attività (26) si esegue subito una `read` di cui si controlla il valore di ritorno; se questo è negativo si è avuto un errore e pertanto si esce immediatamente segnalandolo (27-30), se è nullo significa che il server ha chiuso la connessione, e di nuovo si esce con stampando prima un messaggio di avviso (31-34), altrimenti (35-39) si effettua la terminazione della stringa e la si stampa a sullo *standard output*, uscendo in caso di errore, per ripetere il ciclo da capo.

Con questo meccanismo il programma invece di essere bloccato in lettura sullo *standard input* resta bloccato sulla `select`, che ritorna soltanto quando viene rilevata attività su uno dei due file descriptor posti sotto controllo. Questo di norma avviene solo quando si è scritto qualcosa sullo *standard input*, o quando si riceve dal socket la risposta a quanto si era appena scritto.

Ma adesso il client diventa capace di accorgersi immediatamente della terminazione del server; in tal caso infatti il server chiuderà il socket connesso, ed alla ricezione del FIN la funzione `select` ritornerà (come illustrato in sez. 15.6.1) segnalando una condizione di end of file, per cui il nostro client potrà uscire immediatamente.

Riprendiamo la situazione affrontata in sez. 15.5.2, terminando il server durante una connessione, in questo caso quello che otterremo, una volta scritta una prima riga ed interrotto il server con un C-c, sarà:

```
[piccardi@gont sources]$ ./echo 192.168.1.1
Prima riga
Prima riga
EOF sul socket
```

dove l'ultima riga compare immediatamente dopo aver interrotto il server. Il nostro client infatti è in grado di accorgersi immediatamente che il socket connesso è stato chiuso ed uscire immediatamente.

Veniamo allora agli altri scenari di terminazione anomala visti in sez. 15.5.3. Il primo di questi è l'interruzione fisica della connessione; in questo caso avremo un comportamento

analogo al precedente, in cui si scrive una riga e non si riceve risposta dal server e non succede niente fino a quando non si riceve un errore di EHOSTUNREACH o ETIMEDOUT a seconda dei casi.

La differenza è che stavolta potremo scrivere più righe dopo l'interruzione, in quanto il nostro client dopo aver inviato i dati non si bloccherà più nella lettura dal socket, ma nella **select**; per questo potrà accettare ulteriore dati che scriverà di nuovo sul socket, fintanto che c'è spazio sul buffer di uscita (ecceduto il quale si bloccherà in scrittura).

Si ricordi infatti che il client non ha modo di determinare se la connessione è attiva o meno (dato che in molte situazioni reali l'inattività può essere temporanea). Tra l'altro se si ricollega la rete prima della scadenza del timeout, potremo anche verificare come tutto quello che si era scritto viene poi effettivamente trasmesso non appena la connessione ridiventa attiva, per cui otterremo qualcosa del tipo:

```
[piccardi@gont sources]$ ./echo 192.168.1.1
Prima riga
Prima riga
Seconda riga dopo l'interruzione
Terza riga
Quarta riga
Seconda riga dopo l'interruzione
Terza riga
Quarta riga
```

in cui, una volta riconnessa la rete, tutto quello che abbiamo scritto durante il periodo di disconnessione restituito indietro e stampato immediatamente.

Lo stesso comportamento visto in sez. 15.5.2 si riottiene nel caso di un crollo completo della macchina su cui sta il server. In questo caso di nuovo il client non è in grado di accorgersi di niente dato che si suppone che il programma server non venga terminato correttamente, ma si blocchi tutto senza la possibilità di avere l'emissione di un segmento FIN che segnala la terminazione della connessione.

Di nuovo fintanto che la connessione non si riattiva (con il riavvio della macchina del server) il client non è in grado di fare altro che accettare dell'input e tentare di inviarlo. La differenza in questo caso è che non appena la connessione ridiventa attiva i dati verranno sì trasmessi, ma essendo state perse tutte le informazioni relative alle precedenti connessioni ai tentativi di scrittura del client sarà risposto con un segmento RST che provocherà il ritorno di **select** per la ricezione di un errore di ECONNRESET.

15.6.3 La funzione shutdown

Come spiegato in sez. 15.1.3 il procedimento di chiusura di un socket TCP prevede che da entrambe le parti venga emesso un segmento FIN. È pertanto del tutto normale dal punto di vista del protocollo che uno dei due capi chiuda la connessione quando l'altro capo la lascia aperta; abbiamo incontrato questa situazione nei vari scenari critici di sez. 15.5.

È pertanto possibile avere una situazione in cui un capo della connessione, non avendo più nulla da scrivere, possa chiudere il socket, segnalando così l'avvenuta terminazione della trasmissione (l'altro capo riceverà infatti un *end-of-file* in lettura) mentre dall'altra parte si potrà proseguire la trasmissione dei dati scrivendo sul socket che da quel lato è ancora aperto. Questa è quella situazione in cui si dice che il socket è *mezzo chiuso* ("half closed").

Il problema che si pone è che se la chiusura del socket è effettuata con la funzione **close**, come spiegato in sez. 15.2.6, si perde ogni possibilità di poter leggere quanto l'altro capo può star continuando a scrivere. Per permettere di segnalare che si è finito con la scrittura, continuando al contempo a leggere quanto può provenire dall'altro capo del socket, si può usare la funzione **shutdown**, il cui prototipo è:

```
#include <sys/socket.h>
int shutdown(int sockfd, int how)
```

Chiude un lato della connessione fra due socket.

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

EBADF sockfd non è un file descriptor valido.
 EINVAL il valore di how non è valido.
 ENOTCONN il socket non è connesso.
 ENOTSOCK il file descriptor non corrisponde a un socket.

La funzione prende come primo argomento il socket `sockfd` su cui si vuole operare e come secondo argomento un valore intero `how` che indica la modalità di chiusura del socket, quest'ultima può prendere soltanto tre valori:

- SHUT_RD** chiude il lato in lettura del socket, non sarà più possibile leggere dati da esso, tutti gli eventuali dati trasmessi dall'altro capo del socket saranno automaticamente scartati dal kernel, che, in caso di socket TCP, provvederà comunque ad inviare i relativi segmenti di ACK.
- SHUT_WR** chiude il lato in scrittura del socket, non sarà più possibile scrivere dati su di esso. Nel caso di socket TCP la chiamata causa l'emissione di un segmento FIN, secondo la procedura chiamata *half-close*. Tutti i dati presenti nel buffer di scrittura prima della chiamata saranno inviati, seguiti dalla sequenza di chiusura illustrata in sez. 15.1.3.
- SHUT_RDWR** chiude sia il lato in lettura che quello in scrittura del socket. È equivalente alla chiamata in sequenza con SHUT_RD e SHUT_WR.

Ci si può chiedere quale sia l'utilità di avere introdotto SHUT_RDWR quando questa sembra rendere `shutdown` del tutto equivalente ad una `close`. In realtà non è così, esiste infatti un'altra differenza con `close`, più sottile. Finora infatti non ci siamo presi la briga di sottolineare in maniera esplicita che, come per i file e le *fifo*, anche per i socket possono esserci più riferimenti contemporanei ad uno stesso socket.

Per cui si avrebbe potuto avere l'impressione che sia una corrispondenza univoca fra un socket ed il file descriptor con cui vi si accede. Questo non è assolutamente vero, (e lo abbiamo già visto nel codice del server di fig. 15.13), ed è invece assolutamente normale che, come per gli altri oggetti, ci possano essere più file descriptor che fanno riferimento allo stesso socket.

Allora se avviene uno di questi casi quello che succederà è che la chiamata a `close` darà effettivamente avvio alla sequenza di chiusura di un socket soltanto quando il numero di riferimenti a quest'ultimo diventerà nullo. Fintanto che ci sono file descriptor che fanno riferimento ad un socket l'uso di `close` si limiterà a deallocare nel processo corrente il file descriptor utilizzato, ma il socket resterà pienamente accessibile attraverso tutti gli altri riferimenti.

Se torniamo all'esempio originale del server di fig. 15.13 abbiamo infatti che ci sono due `close`, una sul socket connesso nel padre, ed una sul socket in ascolto nel figlio, ma queste non effettuano nessuna chiusura reale di detti socket, dato che restano altri riferimenti attivi, uno al socket connesso nel figlio ed uno a quello in ascolto nel padre.

Questo non avviene affatto se si usa `shutdown` con argomento SHUT_RDWR al posto di `close`; in questo caso infatti la chiusura del socket viene effettuata immediatamente, indipendentemente dalla presenza di altri riferimenti attivi, e pertanto sarà efficace anche per tutti gli altri file descriptor con cui, nello stesso o in altri processi, si fa riferimento allo stesso socket.

Il caso più comune di uso di `shutdown` è comunque quello della chiusura del lato in scrittura, per segnalare all'altro capo della connessione che si è concluso l'invio dei dati, restando

comunque in grado di ricevere quanto questi potrà ancora inviarci. Questo è ad esempio l'uso che ci serve per rendere finalmente completo il nostro esempio sul servizio *echo*.

Il nostro client infatti presenta ancora un problema, che nell'uso che finora ne abbiamo fatto non è emerso, ma che ci aspetta dietro l'angolo non appena usciamo dall'uso interattivo e proviamo ad eseguirlo redirigendo *standard input* e *standard output*. Così se eseguiamo:

```
[piccardi@gont sources]$ ./echo 192.168.1.1 < ../fileadv.tex > copia
```

vedremo che il file *copia* risulta mancare della parte finale.

Per capire cosa avviene in questo caso occorre tenere presente come avviene la comunicazione via rete; quando redirigiamo lo *standard input* il nostro client inizierà a leggere il contenuto del file *../fileadv.tex* a blocchi di dimensione massima pari a *MAXLINE* per poi scriverlo, alla massima velocità consentitagli dalla rete, sul socket. Dato che la connessione è con una macchina remota, occorre un certo tempo perché i pacchetti vi arrivino, vengano processati, e poi tornino indietro. Considerando trascurabile il tempo di processo, questo tempo è quello impiegato nella trasmissione via rete, che viene detto *RTT* (dalla denominazione inglese *Round Trip Time*) ed è quello che viene stimato con l'uso del comando *ping*.

A questo punto, se torniamo al codice mostrato in fig. 15.22, possiamo vedere che mentre i pacchetti sono in transito sulla rete il client continua a leggere e a scrivere fintanto che il file in ingresso finisce. Però non appena viene ricevuto un *end-of-file* in ingresso il nostro client termina. Nel caso interattivo, in cui si inviavano brevi stringhe una alla volta, c'era sempre il tempo di eseguire la lettura completa di quanto il server rimandava indietro. In questo caso invece, quando il client termina, essendo la comunicazione saturata e a piena velocità, ci saranno ancora pacchetti in transito sulla rete che devono arrivare al server e poi tornare indietro, ma siccome il client esce immediatamente dopo la fine del file in ingresso, questi non faranno a tempo a completare il percorso e verranno persi.

Per evitare questo tipo di problema, invece di uscire una volta completata la lettura del file in ingresso, occorre usare *shutdown* per effettuare la chiusura del lato in scrittura del socket. In questo modo il client segnalerà al server la chiusura del flusso dei dati, ma potrà continuare a leggere quanto il server gli sta ancora inviando indietro, fino a quando anch'esso, riconosciuta la chiusura del socket in scrittura da parte del client, effettuerà la chiusura dalla sua parte. Solo alla ricezione della chiusura del socket da parte del server il client potrà essere sicuro della ricezione di tutti i dati e della terminazione effettiva della connessione.

Si è allora riportato in fig. 15.23 la versione finale della nostra funzione *ClientEcho*, in grado di gestire correttamente l'intero flusso di dati fra client e server. Il codice completo del client, comprendente la gestione delle opzioni a riga di comando e le istruzioni per la creazione della connessione, si trova nel file *TCP_echo_fourth.c*, distribuito coi sorgenti allegati alla guida.

La nuova versione è molto simile alla precedente di fig. 15.22; la prima differenza è l'introduzione (7) della variabile *eof*, inizializzata ad un valore nullo, che serve a mantenere traccia dell'avvenuta conclusione della lettura del file in ingresso.

La seconda modifica (12-15) è stata quella di rendere subordinata ad un valore nullo di *eof* l'impostazione del file descriptor set per l'osservazione dello *standard input*. Se infatti il valore di *eof* è non nullo significa che si è già raggiunta la fine del file in ingresso ed è pertanto inutile continuare a tenere sotto controllo lo *standard input* nella successiva (16) chiamata a *select*.

Le maggiori modifiche rispetto alla precedente versione sono invece nella gestione (18-22) del caso in cui la lettura con *fgets* restituisce un valore nullo, indice della fine del file. Questa nella precedente versione causava l'immediato ritorno della funzione; in questo caso prima (19) si imposta opportunamente *eof* ad un valore non nullo, dopo di che (20) si effettua la

```

1 void ClientEcho(FILE * filein, int socket)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread, nwrite;
5     int maxfd;
6     fd_set fset;
7     int eof = 0;
8     /* initialize file descriptor set */
9     FD_ZERO(&fset);
10    maxfd = max(fileno(filein), socket) + 1;
11    while (1) {
12        FD_SET(socket, &fset);          /* set for the socket */
13        if (eof == 0) {
14            FD_SET(fileno(filein), &fset); /* set for the standard input */
15        }
16        select(maxfd, &fset, NULL, NULL, NULL); /* wait for read ready */
17        if (FD_ISSET(fileno(filein), &fset)) { /* if ready on stdin */
18            if (fgets(sendbuff, MAXLINE, filein) == NULL) { /* if no input */
19                eof = 1; /* EOF on input */
20                shutdown(socket, SHUT_WR); /* close write half */
21                FD_CLR(fileno(filein), &fset); /* no more interest on stdin */
22            } else { /* else we have to write to socket */
23                nwrite = FullWrite(socket, sendbuff, strlen(sendbuff));
24                if (nwrite < 0) { /* on error stop */
25                    printf("Errore_in_scrivtura:_%s", strerror(errno));
26                    return;
27                }
28            }
29        }
30        if (FD_ISSET(socket, &fset)) { /* if ready on socket */
31            nread = read(socket, recvbuff, strlen(sendbuff)); /* do read */
32            if (nread < 0) { /* error condition, stop client */
33                printf("Errore_in_lettura:_%s\n", strerror(errno));
34                return;
35            }
36            if (nread == 0) { /* server closed connection, stop */
37                if (eof == 1) {
38                    return;
39                } else {
40                    printf("EOF_prematuro_sul_socket\n");
41                    return;
42                }
43            }
44            recvbuff[nread] = 0; /* else read is ok, write on stdout */
45            if (fputs(recvbuff, stdout) == EOF) {
46                perror("Errore_in_scrivtura_su_terminale");
47                return;
48            }
49        }
50    }
51 }

```

Figura 15.23: La sezione nel codice della versione finale della funzione ClientEcho, che usa shutdown per una conclusione corretta della connessione.

chiusura del lato in scrittura del socket con `shutdown`. Infine (21) si usa la macro `FD_CLR` per togliere lo *standard input* dal *file descriptor set*.

In questo modo anche se la lettura del file in ingresso è conclusa, la funzione non esce dal ciclo principale (11–50), ma continua ad eseguirlo ripetendo la chiamata a `select` per tenere sotto controllo soltanto il socket connesso, dal quale possono arrivare altri dati, che saranno letti (31) ed opportunamente trascritti (44–48) sullo *standard output*.

Il ritorno della funzione, e la conseguente terminazione normale del client, viene invece adesso gestito all'interno (30–49) della lettura dei dati dal socket; se infatti dalla lettura del socket si riceve una condizione di *end-of-file*, la si tratterà (36–43) in maniera diversa a seconda del valore di `eof`. Se infatti questa è diversa da zero (37–39), essendo stata completata la lettura del file in ingresso, vorrà dire che anche il server ha concluso la trasmissione dei dati restanti, e si potrà uscire senza errori, altrimenti si stamperà (40–42) un messaggio di errore per la chiusura precoce della connessione.

15.6.4 Un server basato sull'I/O *multiplexing*

Seguendo di nuovo le orme di Stevens in [?] vediamo ora come con l'utilizzo dell'I/O *multiplexing* diventi possibile riscrivere completamente il nostro server *echo* con una architettura completamente diversa, in modo da evitare di dover creare un nuovo processo tutte le volte che si ha una connessione.²⁵

La struttura del nuovo server è illustrata in fig. 15.24, in questo caso avremo un solo processo che ad ogni nuova connessione da parte di un client sul socket in ascolto si limiterà a registrare l'entrata in uso di un nuovo file descriptor ed utilizzerà `select` per rilevare la presenza di dati in arrivo su tutti i file descriptor attivi, operando direttamente su ciascuno di essi.

Figura 15.24: Schema del nuovo server *echo* basato sull'I/O *multiplexing*.

La sezione principale del codice del nuovo server è illustrata in fig. 15.25. Si è tralasciata al solito la gestione delle opzioni, che è identica alla versione precedente. Resta invariata anche tutta la parte relativa alla gestione dei segnali, degli errori, e della cessione dei privilegi, così come è identica la gestione della creazione del socket (si può fare riferimento al codice già illustrato in sez. 15.4.3); al solito il codice completo del server è disponibile coi sorgenti allegati nel file `select_echo.c`.

In questo caso, una volta aperto e messo in ascolto il socket, tutto quello che ci servirà sarà chiamare `select` per rilevare la presenza di nuove connessioni o di dati in arrivo, e processarli immediatamente. Per realizzare lo schema mostrato in fig. 15.24 il programma usa una tabella dei socket connessi mantenuta nel vettore `fd_open` dimensionato al valore di `FD_SETSIZE`, ed una variabile `max_fd` per registrare il valore più alto dei file descriptor aperti.

Prima di entrare nel ciclo principale (5–53) la nostra tabella viene inizializzata (2) a zero (valore che utilizzeremo come indicazione del fatto che il relativo file descriptor non è aperto), mentre il valore massimo (3) per i file descriptor aperti viene impostato a quello del socket in ascolto, in quanto esso è l'unico file aperto, oltre i tre standard, e pertanto avrà il valore più alto, che verrà anche (4) inserito nella tabella.

La prima sezione (6–8) del ciclo principale esegue la costruzione del *file descriptor set* `fset` in base ai socket connessi in un certo momento; all'inizio ci sarà soltanto il socket in ascolto ma nel prosieguo delle operazioni verranno utilizzati anche tutti i socket connessi registrati

²⁵ne faremo comunque una realizzazione diversa rispetto a quella presentata da Stevens in [?].

```

1  ...
2  memset(fd_open, 0, FD_SETSIZE); /* clear array of open files */
3  max_fd = list_fd;               /* maximum now is listening socket */
4  fd_open[max_fd] = 1;
5  while (1) { /* main loop, wait for connection and data inside a select */
6      FD_ZERO(&fset);             /* clear fd_set */
7      for (i = list_fd; i <= max_fd; i++) /* initialize fd_set */
8          if (fd_open[i] != 0) FD_SET(i, &fset);
9      while ( ((n = select(max_fd + 1, &fset, NULL, NULL, NULL)) < 0)
10             && (errno == EINTR)); /* wait for data or connection */
11      if (n < 0) { /* on real error exit */
12          PrintErr("select_error");
13          exit(1);
14      }
15      if (FD_ISSET(list_fd, &fset)) { /* if new connection */
16          n--; /* decrement active */
17          len = sizeof(c_addr); /* and call accept */
18          if ((fd = accept(list_fd, (struct sockaddr *)&c_addr, &len)) < 0) {
19              PrintErr("accept_error");
20              exit(1);
21          }
22          fd_open[fd] = 1; /* set new connection socket */
23          if (max_fd < fd) max_fd = fd; /* if needed set new maximum */
24      }
25      i = list_fd; /* first socket to look */
26      while (n != 0) { /* loop on open connections */
27          i++; /* start after listening socket */
28          if (fd_open[i] == 0) continue; /* closed, go next */
29          if (FD_ISSET(i, &fset)) { /* if active process it */
30              n--; /* decrease active */
31              nread = read(i, buffer, MAXLINE); /* read operations */
32              if (nread < 0) {
33                  PrintErr("Errore_in_lettura");
34                  exit(1);
35              }
36              if (nread == 0) { /* if closed connection */
37                  close(i); /* close file */
38                  fd_open[i] = 0; /* mark as closed in table */
39                  if (max_fd == i) { /* if was the maximum */
40                      while (fd_open[--i] == 0); /* loop down */
41                      max_fd = i; /* set new maximum */
42                      break; /* and go back to select */
43                  }
44                  continue; /* continue loop on open */
45              }
46              nwrite = FullWrite(i, buffer, nread); /* write data */
47              if (nwrite) {
48                  PrintErr("Errore_in_scrittura");
49                  exit(1);
50              }
51          }
52      }
53  }
54  ...

```

Figura 15.25: La sezione principale della nuova versione di server *echo* basato sull'uso della funzione *select*.

nella tabella `fd_open`. Dato che la chiamata di `select` modifica il valore del *file descriptor set* è necessario ripetere (6) ogni volta il suo azzeramento per poi procedere con il ciclo (7-8) in cui si impostano i socket trovati attivi.

Per far questo si usa la caratteristica dei file descriptor, descritta in sez. 5.1.2, per cui il kernel associa sempre ad ogni nuovo file il file descriptor con il valore più basso disponibile. Questo fa sì che si possa eseguire il ciclo (7) a partire da un valore minimo, che sarà sempre quello del socket in ascolto, mantenuto in `list_fd`, fino al valore massimo di `max_fd` che dovremo aver cura di tenere aggiornato. Dopo di che basterà controllare (8) nella nostra tabella se il file descriptor è in uso o meno,²⁶ e impostare `fset` di conseguenza.

Una volta inizializzato con i socket aperti il nostro *file descriptor set* potremo chiamare `select` per fargli osservare lo stato degli stessi (in lettura, presumendo che la scrittura sia sempre consentita). Come per il precedente esempio di sez. 15.4.6, essendo questa l'unica funzione che può bloccarsi ed essere interrotta da un segnale, la eseguiamo (9-10) all'interno di un ciclo di `while`, che la ripete indefinitamente qualora esca con un errore di `EINTR`. Nel caso invece di un errore normale si provvede (11-14) ad uscire dal programma stampando un messaggio di errore.

Infine quando la funzione ritorna normalmente avremo in `n` il numero di socket da controllare. Nello specifico si danno due casi per cui `select` può ritornare: o si è ricevuta una nuova connessione ed è pronto il socket in ascolto, sul quale si può eseguire `accept`, o c'è attività su uno dei socket connessi, sui quali si può eseguire `read`.

Il primo caso viene trattato immediatamente (15-24): si controlla (15) che il socket in ascolto sia fra quelli attivi, nel qual caso anzitutto (16) se ne decrementa il numero mantenuto nella variabile `n`. Poi, inizializzata (17) la lunghezza della struttura degli indirizzi, si esegue `accept` per ottenere il nuovo socket connesso, controllando che non ci siano errori (18-21). In questo caso non c'è più la necessità di controllare per interruzioni dovute a segnali, in quanto siamo sicuri che `accept` non si bloccherà. Per completare la trattazione occorre a questo punto aggiungere (22) il nuovo file descriptor alla tabella di quelli connessi, ed inoltre, se è il caso, aggiornare (23) il valore massimo in `max_fd`.

Una volta controllato l'arrivo di nuove connessioni si passa a verificare se ci sono dati sui socket connessi, per questo si ripete un ciclo (26-52) fintanto che il numero di socket attivi indicato dalla variabile `n` resta diverso da zero. In questo modo, se l'unico socket con attività era quello connesso, avendola opportunamente decrementata in precedenza, essa risulterà nulla, pertanto il ciclo di verifica verrà saltato e si ritornerà all'inizio del ciclo principale, ripetendo, dopo l'inizializzazione del *file descriptor set* con i nuovi valori nella tabella, la chiamata di `select`.

Se il socket attivo non è quello in ascolto, o ce ne sono comunque anche altri, il valore di `n` non sarà nullo ed il controllo sarà eseguito. Prima di entrare nel ciclo di verifica comunque si inizializza (25) il valore della variabile `i`, che useremo come indice nella tabella `fd_open`, al valore minimo, corrispondente al file descriptor del socket in ascolto.

Il primo passo (27) nella verifica è incrementare il valore dell'indice `i` per posizionarsi sul primo valore possibile per un file descriptor associato ad un eventuale socket connesso, dopo di che si controlla (28) se questo è nella tabella dei socket connessi, chiedendo la ripetizione del ciclo in caso contrario. Altrimenti si passa a verificare (29) se il file descriptor corrisponde ad uno di quelli attivi, e nel caso si esegue (30) una lettura, uscendo con un messaggio in caso di errore (31-35).

²⁶si tenga presente che benché il kernel assegni sempre il primo valore libero, si potranno sempre avere dei *buchi* nella nostra tabella dato che nelle operazioni i socket saranno aperti e chiusi in corrispondenza della creazione e conclusione delle connessioni.

Se (36) il numero di byte letti `nread` è nullo si è in presenza di una *end-of-file*, indice che una connessione che si è chiusa, che deve essere trattata (36–45) opportunamente. Il primo passo è chiudere (37) anche il proprio capo del socket e rimuovere (38) il file descriptor dalla tabella di quelli aperti, inoltre occorre verificare (39) se il file descriptor chiuso è quello con il valore più alto, nel qual caso occorre trovare (39–43) il nuovo massimo, altrimenti (44) si può ripetere il ciclo da capo per esaminare (se ne restano) ulteriori file descriptor attivi.

Se però è stato chiuso il file descriptor più alto, dato che la scansione dei file descriptor attivi viene fatta a partire dal valore più basso, questo significa che siamo anche arrivati alla fine della scansione, per questo possiamo utilizzare direttamente il valore dell'indice `i` con un ciclo all'indietro (40) che trova il primo valore per cui la tabella presenta un file descriptor aperto, e lo imposta (41) come nuovo massimo, per poi tornare (42) al ciclo principale con un `break`, e rieseguire `select`.

Se infine si sono effettivamente letti dei dati dal socket (ultimo caso rimasto) si potrà invocare immediatamente (46) `FullWrite` per riscriverli indietro sul socket stesso, avendo cura di uscire con un messaggio in caso di errore (47–50). Si noti che nel ciclo si esegue una sola lettura, contrariamente a quanto fatto con la precedente versione (si riveda il codice di fig. 15.19) in cui si continuava a leggere fintanto che non si riceveva un *end-of-file*, questo perché usando l'*I/O multiplexing* non si vuole essere bloccati in lettura.

L'uso di `select` ci permette di trattare automaticamente anche il caso in cui la `read` non è stata in grado di leggere tutti i dati presenti sul socket, dato che alla iterazione successiva `select` ritornerà immediatamente segnalando l'ulteriore disponibilità.

Il nostro server comunque soffre di una vulnerabilità per un attacco di tipo *Denial of Service*. Il problema è che in caso di blocco di una qualunque delle funzioni di I/O, non avendo usato processi separati, tutto il server si ferma e non risponde più a nessuna richiesta. Abbiamo scongiurato questa evenienza per l'I/O in ingresso con l'uso di `select`, ma non vale altrettanto per l'I/O in uscita. Il problema pertanto può sorgere qualora una delle chiamate a `write` effettuate da `FullWrite` si blocchi.

Con il funzionamento normale questo non accade in quanto il server si limita a scrivere quanto riceve in ingresso, ma qualora venga utilizzato un client malevolo che esegua solo scritture e non legga mai indietro l'eco del server, si potrebbe giungere alla saturazione del buffer di scrittura, ed al conseguente blocco del server su di una `write`.

Le possibili soluzioni in questo caso sono quelle di ritornare ad eseguire il ciclo di risposta alle richieste all'interno di processi separati, utilizzare un timeout per le operazioni di scrittura, o eseguire queste ultime in modalità non bloccante, concludendo le operazioni qualora non vadano a buon fine.

15.6.5 I/O multiplexing con poll

Finora abbiamo trattato le problematiche risolubili con l'*I/O multiplexing* impiegando la funzione `select`. Questo è quello che avviene nella maggior parte dei casi, in quanto essa è nata sotto BSD proprio per affrontare queste problematiche con i socket. Abbiamo però visto in sez. 10.2 come la funzione `poll` possa costituire una alternativa a `select`, con alcuni vantaggi, non soffrendo delle limitazioni dovute all'uso dei *file descriptor set*.

Ancora una volta in sez. 10.2.3 abbiamo trattato la funzione in maniera generica, parlando di file descriptor, ma come per `select` quando si ha a che fare con dei socket, il concetto di essere *pronti* per l'I/O deve essere specificato nei dettagli per tener conto delle condizioni della rete. Inoltre deve essere specificato come viene classificato il traffico nella suddivisione fra dati normali e prioritari. In generale pertanto:

- i dati inviati su un socket vengono considerati traffico normale, pertanto vengono rilevati alla loro ricezione sull'altro capo da una selezione effettuata con `POLLIN` o `POLLRDNORM`;
- i dati urgenti *out-of-band* (vedi sez. 18.1.4) su un socket TCP vengono considerati traffico prioritario e vengono rilevati da una condizione `POLLIN`, `POLLPRI` o `POLLRDBAND`.
- la chiusura di una connessione (cioè la ricezione di un segmento `FIN`) viene considerato traffico normale, pertanto viene rilevato da una condizione `POLLIN` o `POLLRDNORM`, ma una conseguente chiamata a `read` restituirà 0.
- la disponibilità di spazio sul socket per la scrittura di dati viene segnalata con una condizione `POLLOUT`.
- quando uno dei due capi del socket chiude un suo lato della connessione con `shutdown` si riceve una condizione di `POLLHUP`.
- la presenza di un errore sul socket (sia dovuta ad un segmento `RST` che a timeout) viene considerata traffico normale, ma viene segnalata anche dalla condizione `POLLERR`.
- la presenza di una nuova connessione su un socket in ascolto può essere considerata sia traffico normale che prioritario, nel caso di Linux la realizzazione dello *stack TCP* la classifica come normale.

Come esempio dell'uso di `poll` proviamo allora a riscrivere il server *echo* secondo lo schema di fig. 15.24 usando `poll` al posto di `select`. In questo caso dovremo fare qualche modifica, per tenere conto della diversa sintassi delle due funzioni, ma la struttura del programma resta sostanzialmente la stessa.

In fig. 15.26 è riportata la sezione principale della nuova versione del server, la versione completa del codice è riportata nel file `poll_echo.c` dei sorgenti allegati alla guida. Al solito nella figura si sono tralasciate la gestione delle opzioni, la creazione del socket in ascolto, la cessione dei privilegi e le operazioni necessarie a far funzionare il programma come demone, privilegiando la sezione principale del programma.

Come per il precedente server basato su `select` il primo passo (2-8) è quello di inizializzare le variabili necessarie. Dato che in questo caso dovremo usare un vettore di strutture occorre anzitutto (2) allocare la memoria necessaria utilizzando il numero massimo `n` di socket osservabili, che viene impostato attraverso l'opzione `-n` ed ha un valore di default di 256.

Dopo di che si preimposta (3) il valore `max_fd` del file descriptor aperto con valore più alto a quello del socket in ascolto (al momento l'unico), e si provvede (4-7) ad inizializzare le strutture, disabilitando l'osservazione (5) con un valore negativo del campo `fd`, ma predisponendo (6) il campo `events` per l'osservazione dei dati normali con `POLLRDNORM`. Infine (8) si attiva l'osservazione del socket in ascolto inizializzando la corrispondente struttura. Questo metodo comporta, in modalità interattiva, lo spreco di tre strutture (quelle relative a *standard input*, *standard output* e *standard error*) che non vengono mai utilizzate in quanto la prima è sempre quella relativa al socket in ascolto.

Una volta completata l'inizializzazione tutto il lavoro viene svolto all'interno del ciclo principale (9-53) che ha una struttura sostanzialmente identica a quello usato per il precedente esempio basato su `select`. La prima istruzione (10) è quella di eseguire `poll` all'interno di un ciclo che la ripete qualora venisse interrotta da un segnale, da cui si esce soltanto quando la funzione ritorna restituendo nella variabile `n` il numero di file descriptor trovati attivi. Qualora invece si sia ottenuto un errore si procede (11-14) alla terminazione immediata del processo provvedendo a stampare una descrizione dello stesso.

```

1 ...
2 poll_set = (struct pollfd *) malloc(n * sizeof(struct pollfd));
3 max_fd = list_fd; /* maximum now is listening socket */
4 for (i=0; i<n; i++) { /* initialize poll set */
5     poll_set[i].fd = -1;
6     poll_set[i].events = POLLRDNORM;
7 }
8 poll_set[max_fd].fd = list_fd;
9 while (1) { /* main loop, wait for connection and data inside a poll */
10     while ( ((n=poll(poll_set, max_fd + 1, -1)) < 0) && (errno == EINTR));
11     if (n < 0) { /* on real error exit */
12         PrintErr("poll_error");
13         exit(1);
14     }
15     if (poll_set[list_fd].revents & POLLRDNORM) { /* if new connection */
16         n--; /* decrement active */
17         len = sizeof(c_addr); /* and call accept */
18         if ((fd = accept(list_fd, (struct sockaddr *)&c_addr, &len)) < 0) {
19             PrintErr("accept_error");
20             exit(1);
21         }
22         poll_set[fd].fd = fd; /* set new connection socket */
23         if (max_fd < fd) max_fd = fd; /* if needed set new maximum */
24     }
25     i = list_fd; /* first socket to look */
26     while (n != 0) { /* loop until active */
27         i++; /* start after listening socket */
28         if (poll_set[i].fd == -1) continue; /* closed, go next */
29         if (poll_set[i].revents & (POLLRDNORM|POLLERR)) {
30             n--; /* decrease active */
31             nread = read(i, buffer, MAXLINE); /* read operations */
32             if (nread < 0) {
33                 PrintErr("Errore_in_lettura");
34                 exit(1);
35             }
36             if (nread == 0) { /* if closed connection */
37                 close(i); /* close file */
38                 poll_set[i].fd = -1; /* mark as closed in table */
39                 if (max_fd == i) { /* if was the maximum */
40                     while (poll_set[--i].fd == -1); /* loop down */
41                     max_fd = i; /* set new maximum */
42                     break; /* and go back to select */
43                 }
44                 continue; /* continue loop on open */
45             }
46             nwrite = FullWrite(i, buffer, nread); /* write data */
47             if (nwrite) {
48                 PrintErr("Errore_in_scrittura");
49                 exit(1);
50             }
51         }
52     }
53 }
54 ...

```

Figura 15.26: La sezione principale della nuova versione di server *echo* basato sull'uso della funzione `poll`.

Una volta ottenuta dell'attività su un file descriptor si hanno di nuovo due possibilità. La prima è che ci sia attività sul socket in ascolto, indice di una nuova connessione, nel qual caso si controlla (17) se il campo `revents` della relativa struttura è attivo; se è così si provvede (16) a decrementare la variabile `n` (che assume il significato di numero di file descriptor attivi rimasti da controllare) per poi (17-21) effettuare la chiamata ad `accept`, terminando il processo in caso di errore. Se la chiamata ad `accept` ha successo si procede attivando (22) la struttura relativa al nuovo file descriptor da essa ottenuto, modificando (23) infine quando necessario il valore massimo dei file descriptor aperti mantenuto in `max_fd`.

La seconda possibilità è che vi sia dell'attività su uno dei socket aperti in precedenza, nel qual caso si inizializza (25) l'indice `i` del vettore delle strutture `pollfd` al valore del socket in ascolto, dato che gli ulteriori socket aperti avranno comunque un valore superiore. Il ciclo (26-52) prosegue fintanto che il numero di file descriptor attivi, mantenuto nella variabile `n`, è diverso da zero. Se pertanto ci sono ancora socket attivi da individuare si comincia con l'incrementare (27) l'indice e controllare (28) se corrisponde ad un file descriptor in uso analizzando il valore del campo `fd` della relativa struttura e chiudendo immediatamente il ciclo qualora non lo sia. Se invece il file descriptor è in uso si verifica (29) se c'è stata attività controllando il campo `revents`.

Di nuovo se non si verifica la presenza di attività il ciclo si chiude subito, altrimenti si provvederà (30) a decrementare il numero `n` di file descriptor attivi da controllare e ad eseguire (31) la lettura, ed in caso di errore (32-35) al solito lo si notificherà uscendo immediatamente. Qualora invece si ottenga una condizione di *end-of-file* (36-45) si provvederà a chiudere (37) anche il nostro capo del socket e a marcarlo (38) come inutilizzato nella struttura ad esso associata. Infine dovrà essere ricalcolato (39-43) un eventuale nuovo valore di `max_fd`. L'ultimo passo è chiudere (44) il ciclo in quanto in questo caso non c'è più niente da riscrivere all'indietro sul socket.

Se invece si sono letti dei dati si provvede (46) ad effettuarne la riscrittura all'indietro, con il solito controllo ed eventuale uscita e notifica in caso di errore (47-51).

Come si può notare la logica del programma è identica a quella vista in fig. 15.25 per l'analogo server basato su `select`; la sola differenza significativa è che in questo caso non c'è bisogno di rigenerare i *file descriptor set* in quanto l'uscita è indipendente dai dati in ingresso. Si applicano comunque anche a questo server le considerazioni finali di sez. 15.6.4.

Capitolo 16

La gestione dei socket

Esamineremo in questo capitolo una serie di funzionalità aggiuntive relative alla gestione dei socket, come la gestione della risoluzione di nomi e indirizzi, le impostazioni delle varie proprietà ed opzioni relative ai socket, e le funzioni di controllo che permettono di modificarne il comportamento.

16.1 La risoluzione dei nomi

Negli esempi dei capitoli precedenti abbiamo sempre identificato le singole macchine attraverso indirizzi numerici, sfruttando al più le funzioni di conversione elementare illustrate in sez. 14.3 che permettono di passare da un indirizzo espresso in forma *dotted decimal* ad un numero. Vedremo in questa sezione le funzioni utilizzate per poter utilizzare dei nomi simbolici al posto dei valori numerici, e viceversa quelle che permettono di ottenere i nomi simbolici associati ad indirizzi, porte o altre proprietà del sistema.

16.1.1 La struttura del *resolver*

La risoluzione dei nomi è associata tradizionalmente al servizio del *Domain Name Service* che permette di identificare le macchine su internet invece che per numero IP attraverso il relativo *nome a dominio*.¹ In realtà per DNS si intendono spesso i server che forniscono su internet questo servizio, mentre nel nostro caso affronteremo la problematica dal lato client, di un qualunque programma che necessita di compiere questa operazione.

Figura 16.1: Schema di funzionamento delle funzioni del *resolver*.

Inoltre quella fra nomi a dominio e indirizzi IP non è l'unica corrispondenza possibile fra nomi simbolici e valori numerici, come abbiamo visto anche in sez. 6.2.1 per le corrispondenze fra nomi di utenti e gruppi e relativi identificatori numerici; per quanto riguarda però tutti i nomi associati a identificativi o servizi relativi alla rete il servizio di risoluzione è gestito in maniera unificata da un insieme di funzioni fornite con le librerie del C, detto appunto *resolver*.

¹non staremo ad entrare nei dettagli della definizione di cosa è un nome a dominio, dandolo per noto, una introduzione alla problematica si trova in [?] (cap. 9) mentre per una trattazione approfondita di tutte le problematiche relative al DNS si può fare riferimento a [?].

Lo schema di funzionamento del *resolver* è illustrato in fig. 16.1; in sostanza i programmi hanno a disposizione un insieme di funzioni di libreria con cui chiamano il *resolver*, indicate con le frecce nere. Ricevuta la richiesta è quest'ultimo che, sulla base della sua configurazione, esegue le operazioni necessarie a fornire la risposta, che possono essere la lettura delle informazioni mantenute nei relativi dei file statici presenti sulla macchina, una interrogazione ad un DNS (che a sua volta, per il funzionamento del protocollo, può interrogarne altri) o la richiesta ad altri server per i quali sia fornito il supporto, come LDAP.²

La configurazione del *resolver* attiene più alla amministrazione di sistema che alla programmazione, ciò non di meno, prima di trattare le varie funzioni di librerie utilizzate dai programmi, vale la pena fare una panoramica generale. Originariamente la configurazione del *resolver* riguardava esclusivamente le questioni relative alla gestione dei nomi a dominio, e prevedeva solo l'utilizzo del DNS e del file statico `/etc/hosts`.

Per questo aspetto il file di configurazione principale del sistema è `/etc/resolv.conf` che contiene in sostanza l'elenco degli indirizzi IP dei server DNS da contattare; a questo si affiancava (fino alla *glibc* 2.4) il file `/etc/host.conf` il cui scopo principale era indicare l'ordine in cui eseguire la risoluzione dei nomi (se usare prima i valori di `/etc/hosts` o quelli del DNS). Tralasciamo i dettagli relativi alle varie direttive che possono essere usate in questi file, che si trovano nelle rispettive pagine di manuale.

Con il tempo però è divenuto possibile fornire diversi sostituti per l'utilizzo delle associazione statiche in `/etc/hosts`, inoltre oltre alla risoluzione dei nomi a dominio ci sono anche altri nomi da risolvere, come quelli che possono essere associati ad una rete (invece che ad una singola macchina) o ai gruppi di macchine definiti dal servizio NIS,³ o come quelli dei protocolli e dei servizi che sono mantenuti nei file statici `/etc/protocols` e `/etc/services`.

Molte di queste informazioni non si trovano su un DNS, ma in una rete locale può essere molto utile centralizzare il mantenimento di alcune di esse su opportuni server. Inoltre l'uso di diversi supporti possibili per le stesse informazioni (ad esempio il nome delle macchine può essere mantenuto sia tramite `/etc/hosts`, che con il DNS, che con NIS) comporta il problema dell'ordine in cui questi vengono interrogati. Con le implementazioni classiche i vari supporti erano introdotti modificando direttamente le funzioni di libreria, prevedendo un ordine di interrogazione predefinito e non modificabile (a meno di una ricompilazione delle librerie stesse).

Per risolvere questa serie di problemi la risoluzione dei nomi a dominio eseguita dal *resolver* è stata inclusa all'interno di un meccanismo generico per la risoluzione di corrispondenze fra nomi ed informazioni ad essi associate chiamato *Name Service Switch*, cui abbiamo accennato anche in sez. 6.2.1 per quanto riguarda la gestione dei dati associati a utenti e gruppi. Il sistema è stato introdotto la prima volta nella libreria standard di Solaris e la *glibc* ha ripreso lo stesso schema; si tenga presente che questo sistema non esiste per altre librerie standard come la *libc5* o la *uClib*.

Il *Name Service Switch* (cui spesso si fa riferimento con l'acronimo NSS) è un sistema di librerie dinamiche che permette di definire in maniera generica sia i supporti su cui mantenere i dati di corrispondenza fra nomi e valori numerici, sia l'ordine in cui effettuare le ricerche

²la sigla LDAP fa riferimento ad un protocollo, il *Lightweight Directory Access Protocol*, che prevede un meccanismo per la gestione di *elenchi* di informazioni via rete; il contenuto di un elenco può essere assolutamente generico, e questo permette il mantenimento dei più vari tipi di informazioni su una infrastruttura di questo tipo.

³il *Network Information Service* è un servizio, creato da Sun, e poi diffuso su tutte le piattaforme unix-like, che permette di raggruppare all'interno di una rete (in quelli che appunto vengono chiamati *netgroup*) varie macchine, centralizzando i servizi di definizione di utenti e gruppi e di autenticazione, oggi è sempre più spesso sostituito da LDAP.

sui vari supporti disponibili. Il sistema prevede una serie di possibili classi di corrispondenza, quelle attualmente definite sono riportate in tab. 16.1.

Classe	Tipo di corrispondenza
passwd	Corrispondenze fra nome dell'utente e relative proprietà (<i>UID</i> , gruppo principale, ecc.).
shadow	Corrispondenze fra username e password dell'utente (e altre informazioni relative alle password).
group	Corrispondenze fra nome del gruppo e proprietà dello stesso.
aliases	Alias per la posta elettronica.
ethers	Corrispondenze fra numero IP e MAC address della scheda di rete.
hosts	Corrispondenze fra nome a dominio e numero IP.
netgroup	Corrispondenze fra gruppo di rete e macchine che lo compongono.
networks	Corrispondenze fra nome di una rete e suo indirizzo IP.
protocols	Corrispondenze fra nome di un protocollo e relativo numero identificativo.
rpc	Corrispondenze fra nome di un servizio RPC e relativo numero identificativo.
publickey	Chiavi pubbliche e private usate per gli RFC sicuri, utilizzate da NFS e NIS+.
services	Corrispondenze fra nome di un servizio e numero di porta.

Tabella 16.1: Le diverse classi di corrispondenze definite all'interno del *Name Service Switch*.

Il sistema del *Name Service Switch* è controllato dal contenuto del file `/etc/nsswitch.conf`; questo contiene una riga di configurazione per ciascuna di queste classi, che viene iniziata col nome di tab. 16.1 seguito da un carattere “:” e prosegue con la lista dei servizi su cui le relative informazioni sono raggiungibili, scritti nell'ordine in cui si vuole siano interrogati. Pertanto nelle versioni recenti delle librerie è questo file e non `/etc/host.conf` a indicare l'ordine con cui si esegue la risoluzione dei nomi.

Ogni servizio è specificato a sua volta da un nome, come `file`, `dns`, `db`, ecc. che identifica la libreria dinamica che realizza l'interfaccia con esso. Per ciascun servizio se `NAME` è il nome utilizzato dentro `/etc/nsswitch.conf`, dovrà essere presente (usualmente in `/lib`) una libreria `libnss_NAME` che ne implementa le funzioni.

In ogni caso, qualunque sia la modalità con cui ricevono i dati o il supporto su cui vengono mantenuti, e che si usino o meno funzionalità aggiuntive fornite dal sistema del *Name Service Switch*, dal punto di vista di un programma che deve effettuare la risoluzione di un nome a dominio, tutto quello che conta sono le funzioni classiche che il *resolver* mette a disposizione (è cura della *glibc* tenere conto della presenza del *Name Service Switch*) e sono queste quelle che tratteremo nelle sezioni successive.

16.1.2 Le funzioni di interrogazione del DNS

Prima di trattare le funzioni usate normalmente nella risoluzione dei nomi a dominio conviene trattare in maniera più dettagliata il servizio DNS. Come accennato questo, benché esso in teoria sia solo uno dei possibili supporti su cui mantenere le informazioni, in pratica costituisce il meccanismo principale con cui vengono risolti i nomi a dominio. Inoltre esso può fornire anche ulteriori informazioni oltre relative alla risoluzione dei nomi a dominio. Per questo motivo esistono una serie di funzioni di libreria che servono specificamente ad eseguire delle interrogazioni verso un server DNS, funzioni che poi vengono utilizzate anche per realizzare le funzioni generiche di libreria usate dal sistema del *resolver*.

Il sistema del DNS è in sostanza di un database distribuito organizzato in maniera gerarchica, i dati vengono mantenuti in tanti server distinti ciascuno dei quali si occupa della risoluzione del proprio *dominio*; i nomi a dominio sono organizzati in una struttura ad albero analoga a quella dell'albero dei file, con domini di primo livello (come i *.org*), secondo livello (come *.truelite.it*), ecc. In questo caso le separazioni sono fra i vari livelli sono definite dal carattere “.” ed i nomi devono essere risolti da destra verso sinistra.⁴ Il meccanismo funziona con il criterio della *delegazione*, un server responsabile per un dominio di primo livello può delegare la risoluzione degli indirizzi per un suo dominio di secondo livello ad un altro server, il quale a sua volta potrà delegare la risoluzione di un eventuale sotto-dominio di terzo livello ad un altro server ancora.

Come accennato un server DNS è in grado di fare molto altro rispetto alla risoluzione di un nome a dominio in un indirizzo IP: ciascuna voce nel database viene chiamata *resource record*, e può contenere diverse informazioni. In genere i *resource record* vengono classificati per la *classe di indirizzi* cui i dati contenuti fanno riferimento, e per il *tipo* di questi ultimi (ritroveremo classi di indirizzi e tipi di record più avanti in tab. 16.3 e tab. 16.4). Oggigiorno i dati mantenuti nei server DNS sono quasi esclusivamente relativi ad indirizzi internet, per cui in pratica viene utilizzata soltanto una classe di indirizzi; invece le corrispondenze fra un nome a dominio ed un indirizzo IP sono solo uno fra i vari tipi di informazione che un server DNS fornisce normalmente.

L'esistenza di vari tipi di informazioni è un altro dei motivi per cui il *resolver* prevede, oltre a quelle relative alla semplice risoluzione dei nomi, un insieme di funzioni specifiche dedicate all'interrogazione di un server DNS, tutte nella forma *res_nome*. La prima di queste funzioni è *res_init*, il cui prototipo è:

```
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
int res_init(void)
```

Inizializza il sistema del *resolver*.

La funzione ritorna 0 in caso di successo e -1 per un errore.

La funzione legge il contenuto dei file di configurazione per impostare il dominio di default, gli indirizzi dei server DNS da contattare e l'ordine delle ricerche; se non sono specificati server verrà utilizzato l'indirizzo locale, e se non è definito un dominio di default sarà usato quello associato con l'indirizzo locale (ma questo può essere sovrascritto con l'uso della variabile di ambiente *LOCALDOMAIN*). In genere non è necessario eseguire questa funzione esplicitamente, in quanto viene automaticamente chiamata la prima volta che si esegue una qualunque delle altre.

Le impostazioni e lo stato del *resolver* inizializzati da *res_init* vengono mantenuti in una serie di variabili raggruppate nei campi di una apposita struttura. Questa struttura viene definita in *resolv.h* e mantenuta nella variabile globale *_res*, che viene utilizzata internamente da tutte le funzioni dell'interfaccia. Questo consente anche di accedere direttamente al contenuto della variabile all'interno di un qualunque programma, una volta che la sia opportunamente dichiarata con:

```
extern struct __res_state _res;
```

Dato che l'uso di una variabile globale rende tutte le funzioni dell'interfaccia classica non rientranti, queste sono state deprecate in favore di una nuova interfaccia in cui esse sono state

⁴per chi si stia chiedendo quale sia la radice di questo albero, cioè l'equivalente di “/”, la risposta è il dominio speciale “.”, che in genere non viene mai scritto esplicitamente, ma che, come chiunque abbia configurato un server DNS sa bene, esiste ed è gestito dai cosiddetti *root DNS* che risolvono i domini di primo livello.

sostituite da altrettante nuove funzioni, il cui nome è ottenuto apponendo una “n” al nome di quella tradizionale (cioè nella forma `res_nnome`). Tutte le nuove funzioni sono identiche alle precedenti, ma hanno un primo argomento aggiuntivo, `statep`, puntatore ad una struttura dello stesso tipo di `_res`. Questo consente di usare una variabile locale per mantenere lo stato del *resolver*, rendendo le nuove funzioni rientranti. In questo caso per poter utilizzare il nuovo argomento occorrerà una opportuna dichiarazione del relativo tipo di dato con:

```
struct __res_state;
typedef struct __res_state *res_state;
```

Così la nuova funzione utilizzata per inizializzare il *resolver* (che come la precedente viene chiamata automaticamente da tutte altre funzioni) è `res_ninit`, ed il suo prototipo è:

```
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
int res_ninit(res_state statep)
Inizializza il sistema del resolver.
```

La funzione ritorna 0 in caso di successo e -1 per un errore.

Indipendentemente da quale versione delle funzioni si usino, tutti i campi della struttura (`_res` o la variabile puntata da `statep`) sono ad uso interno, e vengono usualmente inizializzate da `res_init` o `res_ninit` in base al contenuto dei file di configurazione e ad una serie di valori di default. L'unico campo che può essere utile modificare è `_res.options` (o l'equivalente della variabile puntata da `statep`), una maschera binaria che contiene una serie di bit che esprimono le opzioni che permettono di controllare il comportamento del *resolver*.

Costante	Significato
RES_INIT	Viene attivato se è stata chiamata <code>res_init</code> .
RES_DEBUG	Stampa dei messaggi di debug.
RES_AAONLY	Accetta solo risposte autoritative.
RES_USEVC	Usa connessioni TCP per contattare i server invece che l'usuale UDP.
RES_PRIMARY	Interroga soltanto server DNS primari.
RES_IGNTC	Ignora gli errori di troncamento, non ritenta la richiesta con una connessione TCP.
RES_RECURSE	Imposta il bit che indica che si desidera eseguire una interrogazione ricorsiva.
RES_DEFNAMES	Se attivo <code>res_search</code> aggiunge il nome del dominio di default ai nomi singoli (che non contengono cioè un “.”).
RES_STAYOPEN	Usato con <code>RES_USEVC</code> per mantenere aperte le connessioni TCP fra interrogazioni diverse.
RES_DNSRCH	Se attivo <code>res_search</code> esegue le ricerche di nomi di macchine nel dominio corrente o nei domini ad esso sovrastanti.
RES_INSECURE1	Blocca i controlli di sicurezza di tipo 1.
RES_INSECURE2	Blocca i controlli di sicurezza di tipo 2.
RES_NOALIASES	Blocca l'uso della variabile di ambiente <code>HOSTALIASES</code> .
RES_USE_INET6	Restituisce indirizzi IPv6 con <code>gethostbyname</code> .
RES_ROTATE	Ruota la lista dei server DNS dopo ogni interrogazione.
RES_NOCHECKNAME	Non controlla i nomi per verificarne la correttezza sintattica.
RES_KEEPTSIG	Non elimina i record di tipo TSIG.
RES_BLAST	Effettua un “blast” inviando simultaneamente le richieste a tutti i server; non ancora implementata.
RES_DEFAULT	Combinazione di <code>RES_RECURSE</code> , <code>RES_DEFNAMES</code> e <code>RES_DNSRCH</code> .

Tabella 16.2: Costanti utilizzabili come valori per `_res.options`.

Per utilizzare questa funzionalità per modificare le impostazioni direttamente da programma occorrerà impostare un opportuno valore per questo campo ed invocare esplicitamente `res_init` o `res_ninit`, dopo di che le altre funzioni prenderanno le nuove impostazioni. Le costanti che definiscono i vari bit di questo campo, ed il relativo significato sono illustrate in tab. 16.2; trattandosi di una maschera binaria un valore deve essere espresso con un opportuno OR aritmetico di dette costanti; ad esempio il valore di default delle opzioni, espresso dalla costante `RES_DEFAULT`, è definito come:

```
#define RES_DEFAULT (RES_RECURSE | RES_DEFNAMES | RES_DNSRCH)
```

Non tratteremo il significato degli altri campi non essendovi necessità di modificarli direttamente; gran parte di essi sono infatti impostati dal contenuto dei file di configurazione, mentre le funzionalità controllate da alcuni di esse possono essere modificate con l'uso delle opportune variabili di ambiente come abbiamo visto per `LOCALDOMAIN`. In particolare con `RES_RETRY` si soprasiede il valore del campo `retry` che controlla quante volte viene ripetuto il tentativo di connettersi ad un server DNS prima di dichiarare fallimento; il valore di default è 4, un valore nullo significa bloccare l'uso del DNS. Infine con `RES_TIMEOUT` si soprasiede il valore del campo `retrans` (preimpostato al valore della omonima costante `RES_TIMEOUT` di `resolv.h`) che è il valore preso come base (in numero di secondi) per definire la scadenza di una richiesta, ciascun tentativo di richiesta fallito viene ripetuto raddoppiando il tempo di scadenza per il numero massimo di volte stabilito da `RES_RETRY`.

La funzione di interrogazione principale è `res_query` (`res_nquery` per la nuova interfaccia), che serve ad eseguire una richiesta ad un server DNS per un nome a dominio *completamente specificato* (quello che si chiama FQDN, *Fully Qualified Domain Name*); il loro prototipo è:

```
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
int res_query(const char *dname, int class, int type, unsigned char *answer, int
anslen)
int res_nquery(res_state statep, const char *dname, int class, int type,
unsigned char *answer, int anslen)
```

Esegue una interrogazione al DNS.

Le funzioni ritornano un valore positivo pari alla lunghezza dei dati scritti nel buffer `answer` in caso di successo e `-1` per un errore.

Le funzioni eseguono una interrogazione ad un server DNS relativa al nome da risolvere passato nella stringa indirizzata da `dname`, inoltre deve essere specificata la classe di indirizzi in cui eseguire la ricerca con `class`, ed il tipo di *resource record* che si vuole ottenere con `type`. Il risultato della ricerca verrà scritto nel buffer di lunghezza `anslen` puntato da `answer` che si sarà opportunamente allocato in precedenza.

Una seconda funzione di ricerca analoga a `res_query`, che prende gli stessi argomenti ma che esegue l'interrogazione con le funzionalità aggiuntive previste dalle due opzioni `RES_DEFNAMES` e `RES_DNSRCH`, è `res_search` (`res_nsearch` per la nuova interfaccia), il cui prototipo è:

```
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
int res_search(const char *dname, int class, int type, unsigned char *answer,
int anslen)
int res_nsearch(res_state statep, const char *dname, int class, int type,
unsigned char *answer, int anslen)
```

Esegue una interrogazione al DNS.

Le funzioni ritornano un valore positivo pari alla lunghezza dei dati scritti nel buffer `answer` in caso di successo e `-1` per un errore.

In sostanza la funzione ripete una serie di chiamate a `res_query` (`res_nquery`) aggiungendo al nome contenuto nella stringa `dname` il dominio di default da cercare, fermandosi non appena trova un risultato. Il risultato di entrambe le funzioni viene scritto nel formato opportuno (che sarà diverso a seconda del tipo di record richiesto) nel buffer di ritorno; sarà compito del programma (o di altre funzioni) estrarre i relativi dati, esistono una serie di funzioni interne usate per la scansione di questi dati, per chi fosse interessato una trattazione dettagliata è riportata nel quattordicesimo capitolo di [?].

Le classi di indirizzi supportate da un server DNS sono tre, ma di queste in pratica oggi viene utilizzata soltanto quella degli indirizzi internet; le costanti che identificano dette classi, da usare come valore per l'argomento `class` delle precedenti funzioni, sono riportate in tab. 16.3 (esisteva in realtà anche una classe `C_CSNET` per la omonima rete, ma è stata dichiarata obsoleta).

Costante	Significato
<code>C_IN</code>	Indirizzi internet, in pratica i soli utilizzati oggi.
<code>C_HS</code>	Indirizzi <i>Hesiod</i> , utilizzati solo al MIT, oggi completamente estinti.
<code>C_CHAOS</code>	Indirizzi per la rete <i>Chaosnet</i> , un'altra rete sperimentale nata al MIT.
<code>C_ANY</code>	Indica un indirizzo di classe qualunque.

Tabella 16.3: Costanti identificative delle classi di indirizzi per l'argomento `class` di `res_query`.

Come accennato le tipologie di dati che sono mantenibili su un server DNS sono diverse, ed a ciascuna di essa corrisponde un diverso tipo di *resource record*. L'elenco delle costanti, ripreso dai file di dichiarazione `arpa/nameser.h` e `arpa/nameser_compat.h`, che definiscono i valori che si possono usare per l'argomento `type` per specificare il tipo di *resource record* da richiedere è riportato in tab. 16.4; le costanti (tolto il `T_iniziale`) hanno gli stessi nomi usati per identificare i record nei file di zona di BIND,⁵ e che normalmente sono anche usati come nomi per indicare i record.

L'elenco di tab. 16.4 è quello di *tutti* i *resource record* definiti, con una breve descrizione del relativo significato. Di tutti questi però viene impiegato correntemente solo un piccolo sottoinsieme, alcuni sono obsoleti ed altri fanno riferimento a dati applicativi che non ci interessano non avendo nulla a che fare con la risoluzione degli indirizzi IP, pertanto non entreranno nei dettagli del significato di tutti i *resource record*, ma solo di quelli usati dalle funzioni del *resolver*. Questi sono sostanzialmente i seguenti (per indicarli si è usata la notazione dei file di zona di BIND):

- A viene usato per indicare la corrispondenza fra un nome a dominio ed un indirizzo IPv4; ad esempio la corrispondenza fra `jojo.truelite.it` e l'indirizzo IP `62.48.34.25`.

⁵BIND, acronimo di *Berkley Internet Name Domain*, è una implementazione di un server DNS, ed, essendo utilizzata nella stragrande maggioranza dei casi, fa da riferimento; i dati relativi ad un certo dominio (cioè i suoi *resource record* vengono mantenuti in quelli che sono usualmente chiamati *file di zona*, e in essi ciascun tipo di dominio è identificato da un nome che è appunto identico a quello delle costanti di tab. 16.4 senza il `T_iniziale`.

Costante	Significato
T_A	Indirizzo di una stazione.
T_NS	Server DNS autoritativo per il dominio richiesto.
T_MD	Destinazione per la posta elettronica.
T_MF	Redistributore per la posta elettronica.
T_CNAME	Nome canonico.
T_SOA	Inizio di una zona di autorità.
T_MB	Nome a dominio di una casella di posta.
T_MG	Nome di un membro di un gruppo di posta.
T_MR	Nome di un cambiamento di nome per la posta.
T_NULL	Record nullo.
T_WKS	Servizio noto.
T_PTR	Risoluzione inversa di un indirizzo numerico.
T_HINFO	Informazione sulla stazione.
T_MINFO	Informazione sulla casella di posta.
T_MX	Server cui instradare la posta per il dominio.
T_TXT	Stringhe di testo (libere).
T_RP	Nome di un responsabile (<i>responsible person</i>).
T_AFSDB	Database per una cella AFS.
T_X25	Indirizzo di chiamata per X.25.
T_ISDN	Indirizzo di chiamata per ISDN.
T_RT	Router.
T_NSAP	Indirizzo NSAP.
T_NSAP_PTR	Risoluzione inversa per NSAP (deprecato).
T_SIG	Firma digitale di sicurezza.
T_KEY	Chiave per firma.
T_PX	Corrispondenza per la posta X.400.
T_GPOS	Posizione geografica.
T_AAAA	Indirizzo IPv6.
T_LOC	Informazione di collocazione.
T_NXT	Dominio successivo.
T_EID	Identificatore di punto conclusivo.
T_NIMLOC	Posizionatore <i>nimrod</i> .
T_SRV	Servizio.
T_ATMA	Indirizzo ATM.
T_NAPTR	Puntatore ad una <i>naming authority</i> .
T_TSIG	Firma di transazione.
T_IXFR	Trasferimento di zona incrementale.
T_AXFR	Trasferimento di zona di autorità.
T_MAILB	Trasferimento di record di caselle di posta.
T_MAILA	Trasferimento di record di server di posta.
T_ANY	Valore generico.

Tabella 16.4: Costanti identificative del tipo di record per l'argomento `type` di `res_query`.

- AAAA** viene usato per indicare la corrispondenza fra un nome a dominio ed un indirizzo IPv6; è chiamato in questo modo dato che la dimensione di un indirizzo IPv6 è quattro volte quella di un indirizzo IPv4.
- PTR** per fornire la corrispondenza inversa fra un indirizzo IP ed un nome a dominio ad esso associato si utilizza questo tipo di record (il cui nome sta per *pointer*).
- CNAME** qualora si abbiano più nomi che corrispondono allo stesso indirizzo (come ad esempio `www.truelite.it` e `sources.truelite.it`, che fanno entrambi riferimento alla stessa macchina (nel caso `dodds.truelite.it`) si può usare questo tipo di record per creare degli *alias* in modo da associare un qualunque altro nome al *nome canonico* della macchina (si chiama così quello associato al record A).

Come accennato in caso di successo le due funzioni di richiesta restituiscono il risultato della interrogazione al server, in caso di insuccesso l'errore invece viene segnalato da un valore di ritorno pari a `-1`, ma in questo caso, non può essere utilizzata la variabile `errno` per riportare un codice di errore, in quanto questo viene impostato per ciascuna delle chiamate al sistema utilizzate dalle funzioni del *resolver*, non avrà alcun significato nell'indicare quale parte del procedimento di risoluzione è fallita.

Per questo motivo è stata definita una variabile di errore separata, `h_errno`, che viene utilizzata dalle funzioni del *resolver* per indicare quale problema ha causato il fallimento della risoluzione del nome. Ad essa si può accedere una volta che la si dichiara con:

```
extern int h_errno;
```

ed i valori che può assumere, con il relativo significato, sono riportati in tab. 16.5.

Costante	Significato
HOST_NOT_FOUND	L'indirizzo richiesto non è valido e la macchina indicata è sconosciuta.
NO_ADDRESS	Il nome a dominio richiesto è valido, ma non ha un indirizzo associato ad esso (alternativamente può essere indicato come NO_DATA).
NO_RECOVERY	Si è avuto un errore non recuperabile nell'interrogazione di un server DNS.
TRY_AGAIN	Si è avuto un errore temporaneo nell'interrogazione di un server DNS, si può ritentare l'interrogazione in un secondo tempo.

Tabella 16.5: Valori possibili della variabile `h_errno`.

Insieme alla nuova variabile vengono definite anche delle nuove funzioni per stampare l'errore a video, analoghe a quelle di sez. 6.5.2 per `errno`, ma che usano il valore di `h_errno`; la prima è `herror` ed il suo prototipo è:

```
#include <netdb.h>
void herror(const char *string)
```

Stampa un errore di risoluzione.

La funzione è l'analoga di `perror` e stampa sullo *standard error* un messaggio di errore corrispondente al valore corrente di `h_errno`, a cui viene anteposta la stringa `string` passata come argomento. La seconda funzione è `hstrerror` ed il suo prototipo è:

```
#include <netdb.h>
const char *hstrerror(int err)
```

Restituisce una stringa corrispondente ad un errore di risoluzione.

che, come l'analoga `strerror`, restituisce una stringa con un messaggio di errore già formato, corrispondente al codice passato come argomento (che si presume sia dato da `h_errno`).

16.1.3 La vecchia interfaccia per la risoluzione dei nomi a dominio

La principale funzionalità del *resolver* resta quella di risolvere i nomi a dominio in indirizzi IP, per cui non ci dedicheremo oltre alle funzioni per effettuare delle richieste generiche al DNS ed esamineremo invece le funzioni del *resolver* dedicate specificamente a questo. Tratteremo in questa sezione l'interfaccia tradizionale, che ormai è deprecata, mentre vedremo nella sezione seguente la nuova interfaccia.

La prima funzione dell'interfaccia tradizionale è `gethostbyname` il cui scopo è ottenere l'indirizzo di una stazione noto il suo nome a dominio, il suo prototipo è:

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name)
    Determina l'indirizzo associato ad un nome a dominio.
```

La funzione ritorna il puntatore ad una struttura di tipo `hostent` contenente i dati associati al nome a dominio in caso di successo o un puntatore nullo per un errore.

La funzione prende come argomento una stringa `name` contenente il nome a dominio che si vuole risolvere, in caso di successo i dati ad esso relativi vengono memorizzati in una opportuna struttura `hostent` la cui definizione è riportata in fig. 16.2.

```
struct hostent {
    char    *h_name;           /* official name of host */
    char    **h_aliases;       /* alias list */
    int      h_addrtype;       /* host address type */
    int      h_length;         /* length of address */
    char    **h_addr_list;     /* list of addresses */
}
#define h_addr  h_addr_list[0] /* for backward compatibility */
```

Figura 16.2: La struttura `hostent` per la risoluzione dei nomi a dominio e degli indirizzi IP.

Quando un programma chiama `gethostbyname` e questa usa il DNS per effettuare la risoluzione del nome, è con i valori contenuti nei relativi record che vengono riempite le varie parti della struttura `hostent`. Il primo campo della struttura, `h_name` contiene sempre il *nome canonico*, che nel caso del DNS è appunto il nome associato ad un record A. Il secondo campo della struttura, `h_aliases`, invece è un puntatore ad vettore di puntatori, terminato da un puntatore nullo. Ciascun puntatore del vettore punta ad una stringa contenente uno degli altri possibili nomi associati allo stesso *nome canonico* (quelli che nel DNS vengono inseriti come record di tipo CNAME).

Il terzo campo della struttura, `h_addrtype`, indica il tipo di indirizzo che è stato restituito, e può assumere soltanto i valori `AF_INET` o `AF_INET6`, mentre il quarto campo, `h_length`, indica la lunghezza dell'indirizzo stesso in byte.

Infine il campo `h_addr_list` è il puntatore ad un vettore di puntatori ai singoli indirizzi; il vettore è terminato da un puntatore nullo. Inoltre, come illustrato in fig. 16.2, viene definito il campo `h_addr` come sinonimo di `h_addr_list[0]`, cioè un riferimento diretto al primo indirizzo della lista.

Oltre ai normali nomi a dominio la funzione accetta come argomento `name` anche indirizzi numerici, in formato dotted decimal per IPv4 o con la notazione illustrata in sez. B.2.5 per IPv6. In tal caso `gethostbyname` non eseguirà nessuna interrogazione remota, ma si limiterà a copiare la stringa nel campo `h_name` ed a creare la corrispondente struttura `in_addr` da indirizzare con `h_addr_list[0]`.

Con l'uso di `gethostbyname` si ottengono solo gli indirizzi IPv4, se si vogliono ottenere degli indirizzi IPv6 occorrerà prima impostare l'opzione `RES_USE_INET6` nel campo `_res.options` e poi chiamare `res_init` (vedi sez. 16.1.2) per modificare le opzioni del *resolver*; dato che questo non è molto comodo è stata definita (è una estensione fornita dalla *glibc*, disponibile anche in altri sistemi unix-like) un'altra funzione, `gethostbyname2`, il cui prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname2(const char *name, int af)
    Determina l'indirizzo del tipo scelto associato ad un nome a dominio.
```

La funzione ritorna il puntatore ad una struttura di tipo `hostent` contenente i dati associati al nome a dominio in caso di successo e un puntatore nullo per un errore.

In questo caso la funzione prende un secondo argomento `af` che indica quale famiglia di indirizzi è quella che dovrà essere utilizzata per selezionare i risultati restituiti dalla funzione; i soli valori consentiti sono `AF_INET` o `AF_INET6` per indicare rispettivamente IPv4 e IPv6 (per questo è necessario l'uso di `sys/socket.h`). Per tutto il resto la funzione è identica a `gethostbyname`, ed identici sono i suoi risultati.

Vediamo allora un primo esempio dell'uso delle funzioni di risoluzione, in fig. 16.3 è riportato un estratto del codice di un programma che esegue una semplice interrogazione al *resolver* usando `gethostbyname` e poi ne stampa a video i risultati. Al solito il sorgente completo, che comprende il trattamento delle opzioni ed una funzione per stampare un messaggio di aiuto, è nel file `mygethost.c` dei sorgenti allegati alla guida.

Il programma richiede un solo argomento che specifichi il nome da cercare, senza il quale (15-18) esce con un errore. Dopo di che (20) si limita a chiamare `gethostbyname`, ricevendo il risultato nel puntatore `data`. Questo (21-24) viene controllato per rilevare eventuali errori, nel qual caso il programma esce dopo aver stampato un messaggio con `herror`.

Se invece la risoluzione è andata a buon fine si inizia (25) con lo stampare il nome canonico, dopo di che (26-30) si stampano eventuali altri nomi. Per questo prima (26) si prende il puntatore alla cima della lista che contiene i nomi e poi (27-30) si esegue un ciclo che sarà ripetuto fin tanto che nella lista si troveranno dei puntatori validi per le stringhe dei nomi (si ricordi che la lista viene terminata da un puntatore nullo); prima (28) si stamperà la stringa e poi (29) si provvederà ad incrementare il puntatore per passare al successivo elemento della lista.

Una volta stampati i nomi si passerà a stampare gli indirizzi, il primo passo (31-38) è allora quello di riconoscere il tipo di indirizzo sulla base del valore del campo `h_addrtype`, stampandolo a video. Si è anche previsto di stampare un errore nel caso (che non dovrebbe mai accadere) di un indirizzo non valido.

Infine (39-44) si stamperanno i valori degli indirizzi, di nuovo (39) si inizierà un puntatore alla cima della lista e si eseguirà un ciclo fintanto che questo punterà ad indirizzi validi in maniera analoga a quanto fatto in precedenza per i nomi a dominio. Si noti come, essendo il campo `h_addr_list` un puntatore ad strutture di indirizzi generiche, questo sia ancora di tipo `char **` e si possa riutilizzare lo stesso puntatore usato per i nomi.

Per ciascun indirizzo valido si provvederà (41) a una conversione con la funzione `inet_ntop` (vedi sez. 14.3) passandole gli opportuni argomenti, questa restituirà la stringa da stampare (42) con il valore dell'indirizzo in `buffer`, che si è avuto la cura di dichiarare inizialmente (10) con dimensioni adeguate; dato che la funzione è in grado di tenere conto automaticamente del tipo di indirizzo non ci sono precauzioni particolari da prendere.⁶

Le funzioni illustrate finora hanno un difetto: utilizzano tutte una area di memoria interna per allocare i contenuti della struttura `hostent` e per questo non possono essere rientranti. L'uso della memoria interna inoltre comporta anche che in due successive chiamate i dati potranno essere sovrascritti.

Si tenga presente poi che copiare il contenuto della sola struttura non è sufficiente per salvare tutti i dati, in quanto questa contiene puntatori ad altri dati, che pure possono essere

⁶volendo essere pignoli si dovrebbe controllare lo stato di uscita, lo si è tralasciato per non appesantire il codice, dato che in caso di indirizzi non validi si sarebbe avuto un errore con `gethostbyname`, ma si ricordi che la sicurezza non è mai troppa.

```

1 int main(int argc, char *argv[])
2 {
3     /*
4     * Variables definition
5     */
6     int i;
7     struct hostent *data;
8     char **alias;
9     char *addr;
10    char buffer[INET6_ADDRSTRLEN];
11    ...
12    /*
13    * Main Body
14    */
15    if ((argc - optind) != 1) {
16        printf("Wrong_number_of_arguments_%d\n", argc - optind);
17        usage();
18    }
19    /* get resolution */
20    data = gethostbyname(argv[1]);
21    if (data == NULL) {
22        perror("Errore_di_risoluzione");
23        exit(1);
24    }
25    printf("Canonical_name_%s\n", data->h_name);
26    alias = data->h_aliases;
27    while (*alias != NULL) {
28        printf("Alias_%s\n", *alias);
29        alias++;
30    }
31    if (data->h_addrtype == AF_INET) {
32        printf("Address_are_IPv4\n");
33    } else if (data->h_addrtype == AF_INET6) {
34        printf("Address_are_IPv6\n");
35    } else {
36        printf("Tipo_di_indirizzo_non_valido\n");
37        exit(1);
38    }
39    alias = data->h_addr_list;
40    while (*alias != NULL) {
41        addr = inet_ntop(data->h_addrtype, *alias, buffer, sizeof(buffer));
42        printf("Indirizzo_%s\n", addr);
43        alias++;
44    }
45    exit(0);
46 }

```

Figura 16.3: Esempio di codice per la risoluzione di un indirizzo.

sovrascritti; per questo motivo, se si vuole salvare il risultato di una chiamata, occorrerà eseguire quella che si chiama una *deep copy*.⁷

Per ovviare a questi problemi nella *glibc* sono definite anche delle versioni rientranti delle

⁷si chiama così quella tecnica per cui, quando si deve copiare il contenuto di una struttura complessa (con puntatori che puntano ad altri dati, che a loro volta possono essere puntatori ad altri dati) si deve copiare non solo il contenuto della struttura, ma eseguire una scansione per risolvere anche tutti i puntatori contenuti in essa (e così via se vi sono altre sotto-strutture con altri puntatori) e copiare anche i dati da questi referenziati.

precedenti funzioni, al solito queste sono caratterizzate dall'avere un suffisso `_r`, pertanto avremo le due funzioni `gethostbyname_r` e `gethostbyname2_r` i cui prototipi sono:

```
#include <netdb.h>
#include <sys/socket.h>
int gethostbyname_r(const char *name, struct hostent *ret, char *buf, size_t buflen,
                    struct hostent **result, int *h_errnop)
int gethostbyname2_r(const char *name, int af, struct hostent *ret, char *buf,
                     size_t buflen, struct hostent **result, int *h_errnop)
Versioni rientranti delle funzioni gethostbyname e gethostbyname2.
```

Le funzioni ritornano 0 in caso di successo ed un valore diverso da zero per un errore.

Gli argomenti `name` (e `af` per `gethostbyname2_r`) hanno lo stesso significato visto in precedenza. Tutti gli altri argomenti hanno lo stesso significato per entrambe le funzioni. Per evitare l'uso di variabili globali si dovrà allocare preventivamente una struttura `hostent` in cui ricevere il risultato, passandone l'indirizzo alla funzione nell'argomento `ret`. Inoltre, dato che `hostent` contiene dei puntatori, dovrà essere allocato anche un buffer in cui le funzioni possano scrivere tutti i dati del risultato dell'interrogazione da questi puntati; l'indirizzo e la lunghezza di questo buffer devono essere indicati con gli argomenti `buf` e `buflen`.

Gli ultimi due argomenti vengono utilizzati per avere indietro i risultati come *value result argument*, si deve specificare l'indirizzo della variabile su cui la funzione dovrà salvare il codice di errore con `h_errnop` e quello su cui dovrà salvare il puntatore che si userà per accedere i dati con `result`.

In caso di successo entrambe le funzioni restituiscono un valore nullo, altrimenti restituiscono un valore non nulla e all'indirizzo puntato da `result` sarà salvato un puntatore nullo, mentre a quello puntato da `h_errnop` sarà salvato il valore del codice di errore, dato che per essere rientrante la funzione non può la variabile globale `h_errno`. In questo caso il codice di errore, oltre ai valori di tab. 16.5, può avere anche quello di `ERANGE` qualora il buffer allocato su `buf` non sia sufficiente a contenere i dati, in tal caso si dovrà semplicemente ripetere l'esecuzione della funzione con un buffer di dimensione maggiore.

Una delle caratteristiche delle interrogazioni al servizio DNS è che queste sono normalmente eseguite con il protocollo UDP, ci sono casi in cui si preferisce che vengano usate connessioni permanenti con il protocollo TCP. Per ottenere questo sono previste delle funzioni apposite (si potrebbero impostare direttamente le opzioni di `__res.options`, ma queste funzioni permettono di semplificare la procedura); la prime sono `sethostent` e `endhostent`, il cui prototipo è:

```
#include <netdb.h>
void sethostent(int stayopen)
    Richiede l'uso di connessioni TCP per le interrogazioni ad un server DNS.
void endhostent(void)
    Disattiva l'uso di connessioni TCP per le interrogazioni ad un server DNS.
```

Le funzioni non restituiscono nulla, e non danno errori.

La funzione `sethostent` permette di richiedere l'uso di connessioni TCP per la richiesta dei dati, e che queste restino aperte per successive richieste; il valore dell'argomento `stayopen` indica se attivare questa funzionalità, un valore diverso da zero, che indica una condizione vera in C, attiva la funzionalità. Per disattivare l'uso delle connessioni TCP si può invece usare `endhostent`, e come si vede la funzione è estremamente semplice, non richiedendo nessun argomento.

Infine si può richiedere la risoluzione inversa di un indirizzo IP od IPv6, per ottenerne il nome a dominio ad esso associato, per fare questo si può usare la funzione `gethostbyaddr`, il cui prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyaddr(const char *addr, int len, int type)
```

Richiede la risoluzione inversa di un indirizzo IP.

La funzione ritorna l'indirizzo ad una struttura `hostent` in caso di successo e `NULL` per un errore.

In questo caso l'argomento `addr` dovrà essere il puntatore ad una appropriata struttura contenente il valore dell'indirizzo IP (o IPv6) che si vuole risolvere. L'uso del tipo `char *` per questo argomento è storico, il dato dovrà essere fornito in una struttura `in_addr` per un indirizzo IPv4 ed una struttura `in6_addr` per un indirizzo IPv6.

Si ricordi inoltre, come illustrato in fig. 14.3, che mentre `in_addr` corrisponde in realtà ad un ordinario numero intero a 32 bit (da esprimere comunque in *network order*) non altrettanto avviene per `in6_addr`, pertanto è sempre opportuno inizializzare questi indirizzi con `inet_pton` (vedi sez. 14.3.3).

Nell'argomento `len` se ne dovrà poi specificare la dimensione (rispettivamente 4 o 16), infine l'argomento `type` deve indicare il tipo di indirizzo, e dovrà essere o `AF_INET` o `AF_INET6`.

La funzione restituisce, in caso di successo, un puntatore ad una struttura `hostent`, solo che in questo caso la ricerca viene eseguita richiedendo al DNS un record di tipo PTR corrispondente all'indirizzo specificato. In caso di errore al solito viene usata la variabile `h_errno` per restituire un opportuno codice. In questo caso l'unico campo del risultato che interessa è `h_name` che conterrà il nome a dominio, la funzione comunque inizializza anche il primo campo della lista `h_addr_list` col valore dell'indirizzo passato come argomento.

Dato che `gethostbyaddr` usa un buffer statico, anche di questa funzione esiste una versione rientrante `gethostbyaddr_r` fornita come estensione dalla *glibc*, il cui prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyaddr_r(const void *addr, socklen_t len, int type,
                                struct hostent *ret, char *buf, size_t buflen,
                                struct hostent **result, int *h_errnop);
```

Richiede la risoluzione inversa di un indirizzo IP.

La funzione ritorna 0 in caso di successo e un valore non nullo per un errore.

La funzione prende per gli argomenti `addr`, `len` e `type` gli stessi valori di `gethostbyaddr` con lo stesso significato, gli argomenti successivi vengono utilizzati per restituire i dati, sono identici a quelli già illustrati in per `gethostbyname_r` e `gethostbyname2_r` e devono essere usati allo stesso modo.

Infine lo standard POSIX prevede la presenza della funzione `gethostent`, il cui prototipo è:

```
#include <netdb.h>
struct hostent *gethostent(void)
```

Ottiene la voce successiva nel database dei nomi a dominio.

La funzione ritorna l'indirizzo ad una struttura `hostent` in caso di successo e `NULL` per un errore.

La funzione dovrebbe ritornare (come puntatore alla solita struttura `hostent` allocata internamente) la voce successiva nel database dei nomi a dominio, ma questo ha un significato soltanto quando è relativo alla lettura dei dati da un file come `/etc/hosts` e non per i risultati

del DNS. Nel caso della *glibc* questa viene usata allora solo per la lettura delle voci presenti in quest'ultimo, come avviene anche in altri sistemi unix-like, ed inoltre ignora le voci relative ad indirizzi IPv6.

Della stessa funzione la *glibc* fornisce anche una versione rientrante `gethostent_r`, il cui prototipo è:

```
#include <netdb.h>
struct hostent *gethostent_r(struct hostent *ret, char *buf, size_t buflen,
                             struct hostent **result, int *h_errnop);
                             Ottiene la voce successiva nel database dei nomi a dominio.
```

La funzione ritorna 0 in caso di successo e un valore non nullo per un errore.

La funzione ha lo stesso effetto di `gethostent`; gli argomenti servono a restituire i risultati in maniera rientrante e vanno usati secondo le modalità già illustrate per `gethostbyname_r` e `gethostbyname2_r`.

Dati i limiti delle funzioni `gethostbyname` e `gethostbyaddr` con l'uso di memoria statica che può essere sovrascritta fra due chiamate successive, e per avere sempre la possibilità di indicare esplicitamente il tipo di indirizzi voluto (cosa che non è possibile con `gethostbyname`), è stata successivamente proposta, nell'RFC 2553 un diversa interfaccia con l'introduzione due nuove funzioni di risoluzione,⁸ `getipnodebyname` e `getipnodebyaddr`, i cui prototipi sono:

```
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
struct hostent *getipnodebyname(const char *name, int af, int flags, int *error_num)
                             Richiede la risoluzione di un nome a dominio.
struct hostent *getipnodebyaddr(const void *addr, size_t len, int af, int *error_num)
                             Richiede la risoluzione inversa di un indirizzo IP.
```

Le funzioni ritornano l'indirizzo ad una struttura `hostent` in caso di successo e NULL per un errore.

Entrambe le funzioni supportano esplicitamente la scelta di una famiglia di indirizzi con l'argomento `af` (che può assumere i valori `AF_INET` o `AF_INET6`), e restituiscono un codice di errore (con valori identici a quelli precedentemente illustrati in tab. 16.5) nella variabile puntata da `error_num`. La funzione `getipnodebyaddr` richiede poi che si specifichi l'indirizzo come per `gethostbyaddr` passando anche la lunghezza dello stesso nell'argomento `len`.

La funzione `getipnodebyname` prende come primo argomento il nome da risolvere, inoltre prevede un apposito argomento `flags`, da usare come maschera binaria, che permette di specificarne il comportamento nella risoluzione dei diversi tipi di indirizzi (IPv4 e IPv6); ciascun bit dell'argomento esprime una diversa opzione, e queste possono essere specificate con un OR aritmetico delle costanti riportate in tab. 16.6.

Entrambe le funzioni restituiscono un puntatore ad una struttura `hostent` che contiene i risultati della ricerca, che viene allocata dinamicamente insieme a tutto lo spazio necessario a contenere i dati in essa referenziati; per questo motivo queste funzioni non soffrono dei problemi dovuti all'uso di una sezione statica di memoria presenti con le precedenti `gethostbyname` e `gethostbyaddr`. L'uso di una allocazione dinamica però comporta anche la necessità di disallocare esplicitamente la memoria occupata dai risultati una volta che questi non siano più necessari; a tale scopo viene fornita la funzione `freehostent`, il cui prototipo è:

⁸dette funzioni sono presenti nella *glibc* 2.1.96, ma essendo considerate deprecated (vedi sez. 16.1.4) sono state rimosse nelle versioni successive.

Costante	Significato
AI_V4MAPPED	Usato con AF_INET6 per richiedere una ricerca su un indirizzo IPv4 invece che IPv6; gli eventuali risultati saranno rimappati su indirizzi IPv6.
AI_ALL	Usato con AI_V4MAPPED; richiede sia indirizzi IPv4 che IPv6, e gli indirizzi IPv4 saranno rimappati in IPv6.
AI_ADDRCONFIG	Richiede che una richiesta IPv4 o IPv6 venga eseguita solo se almeno una interfaccia del sistema è associata ad un indirizzo di tale tipo.
AI_DEFAULT	Il valore di default, è equivalente alla combinazione di AI_ADDRCONFIG e di AI_V4MAPPED.

Tabella 16.6: Valori possibili per i bit dell'argomento flags della funzione `getipnodebyname`.

<pre>#include <netdb.h> #include <sys/types.h> #include <sys/socket.h> void freehostent(struct hostent *ip)</pre>	Disalloca una struttura <code>hostent</code> .
La funzione non ritorna nulla, e non da errori.	

La funzione permette di disallocare una struttura `hostent` precedentemente allocata in una chiamata di `getipnodebyname` o `getipnodebyaddr`, e prende come argomento l'indirizzo restituito da una di queste funzioni.

Infine per concludere la nostra panoramica sulle funzioni di risoluzione dei nomi dobbiamo citare le funzioni che permettono di interrogare gli altri servizi di risoluzione dei nomi illustrati in sez. 16.1.1; in generale infatti ci sono una serie di funzioni nella forma `getXXXbyname` e `getXXXbyaddr` (dove XXX indica il servizio) per ciascuna delle informazioni di rete mantenute dal *Name Service Switch* che permettono rispettivamente di trovare una corrispondenza cercando per nome o per numero.

L'elenco di queste funzioni è riportato nelle colonne finali di tab. 16.7, dove le si sono suddivise rispetto al tipo di informazione che forniscono (riportato in prima colonna). Nella tabella si è anche riportato il file su cui vengono ordinariamente mantenute queste informazioni, che però può essere sostituito da un qualunque supporto interno al *Name Service Switch* (anche se usualmente questo avviene solo per la risoluzione degli indirizzi). Ciascuna funzione fa riferimento ad una sua apposita struttura che contiene i relativi dati, riportata in terza colonna.

Informazione	File	Struttura	Funzioni	
indirizzo	/etc/hosts	hostent	gethostbyname	gethostbyaddr
servizio	/etc/services	servent	getservbyname	getservbyport
rete	/etc/networks	netent	getnetbyname	getnetbyaddr
protocollo	/etc/protocols	protoent	getprotobyname	getprotobyaddr

Tabella 16.7: Funzioni di risoluzione dei nomi per i vari servizi del *Name Service Switch* riguardanti la rete.

Delle funzioni di tab. 16.7 abbiamo trattato finora soltanto quelle relative alla risoluzione dei nomi, dato che sono le più usate, e prevedono praticamente da sempre la necessità di rivolgersi ad una entità esterna; per le altre invece, estensioni fornite dal *Name Service Switch* a parte, si fa sempre riferimento ai dati mantenuti nei rispettivi file.

Dopo la risoluzione dei nomi a dominio una delle ricerche più comuni è quella sui nomi dei servizi di rete più comuni (cioè `http`, `smtp`, ecc.) da associare alle rispettive porte. Le due funzioni da utilizzare per questo sono `getservbyname` e `getservbyport`, che permettono

rispettivamente di ottenere il numero di porta associato ad un servizio dato il nome e viceversa; i loro prototipi sono:

```
#include <netdb.h>
struct servent *getservbyname(const char *name, const char *proto)
struct servent *getservbyport(int port, const char *proto)
    Risolvono il nome di un servizio nel rispettivo numero di porta e viceversa.
```

Le funzioni ritornano il puntatore ad una struttura `servent` con i risultati in caso di successo e `NULL` per un errore.

Entrambe le funzioni prendono come ultimo argomento una stringa `proto` che indica il protocollo per il quale si intende effettuare la ricerca (le informazioni mantenute in `/etc/services` infatti sono relative sia alle porte usate su UDP che su TCP, occorre quindi specificare a quale dei due protocolli si fa riferimento) che nel caso di IP può avere come valori possibili solo `udp` o `tcp`;⁹ se si specifica un puntatore nullo la ricerca sarà eseguita su un protocollo qualsiasi.

Il primo argomento è il nome del servizio per `getservbyname`, specificato tramite la stringa `name`, mentre `getservbyport` richiede il numero di porta in `port`. Entrambe le funzioni eseguono una ricerca sul file `/etc/services`¹⁰ ed estraiono i dati dalla prima riga che corrisponde agli argomenti specificati; se la risoluzione ha successo viene restituito un puntatore ad una apposita struttura `servent` contenente tutti i risultati, altrimenti viene restituito un puntatore nullo. Si tenga presente che anche in questo caso i dati vengono mantenuti in una area di memoria statica e che quindi la funzione non è rientrante.

```
struct servent {
    char    *s_name;           /* official service name */
    char    **s_aliases;       /* alias list */
    int     s_port;            /* port number */
    char    *s_proto;          /* protocol to use */
}
```

Figura 16.4: La struttura `servent` per la risoluzione dei nomi dei servizi e dei numeri di porta.

La definizione della struttura `servent` è riportata in fig. 16.4, il primo campo, `s_name` contiene sempre il nome canonico del servizio, mentre `s_aliases` è un puntatore ad un vettore di stringhe contenenti gli eventuali nomi alternativi utilizzabili per identificare lo stesso servizio. Infine `s_port` contiene il numero di porta e `s_proto` il nome del protocollo.

Come riportato in tab. 16.7 ci sono analoghe funzioni per la risoluzione del nome dei protocolli e delle reti; non staremo a descriverle nei dettagli, in quanto il loro uso è molto limitato, esse comunque utilizzano una loro struttura dedicata del tutto analoga alle precedenti: tutti i dettagli relativi al loro funzionamento possono essere trovati nelle rispettive pagine di manuale.

Oltre alle funzioni di ricerca esistono delle ulteriori funzioni che prevedono una lettura sequenziale delle informazioni mantenute nel *Name Service Switch* (in sostanza permettono di leggere i file contenenti le informazioni riga per riga), che sono analoghe a quelle elencate in tab. 6.9 per le informazioni relative ai dati degli utenti e dei gruppi. Nel caso specifico dei servizi avremo allora le tre funzioni `setservernt`, `getservernt` e `endservernt` i cui prototipi sono:

⁹in teoria si potrebbe avere un qualunque protocollo fra quelli citati in `/etc/protocols`, posto che lo stesso supporti il concetto di *porta*, in pratica questi due sono gli unici presenti.

¹⁰il *Name Service Switch* astrae il concetto a qualunque supporto su cui si possano mantenere i suddetti dati.

```
#include <netdb.h>
struct servent *getservent(void)           Legge la voce successiva nel file /etc/services.

void setservent(int stayopen)              Apre il file /etc/services e si posiziona al suo inizio.

void endservent(void)                      Chiude il file /etc/services.
```

Le due funzioni `setservent` e `endservent` non ritornano nulla, `getservent` restituisce il puntatore ad una struttura `servent` in caso di successo e `NULL` per un errore o fine del file.

La prima funzione, `getservent`, legge una singola voce a partire dalla posizione corrente in `/etc/services`, pertanto si può eseguire una lettura sequenziale dello stesso invocandola più volte. Se il file non è aperto provvede automaticamente ad aprirlo, nel qual caso leggerà la prima voce. La seconda funzione, `setservent`, permette di aprire il file `/etc/services` per una successiva lettura, ma se il file è già stato aperto riporta la posizione di lettura alla prima voce del file, in questo modo si può far ricominciare da capo una lettura sequenziale.

Informazione	Funzioni		
indirizzo	sethostent	gethostent	endhostent
servizio	setservent	getservent	endservent
rete	setnetent	getnetent	endnetent
protocollo	setprotoent	getprotoent	endprotoent

Tabella 16.8: Funzioni lettura sequenziale dei dati del *Name Service Switch*.

L'argomento `stayopen` di `setservent`, se diverso da zero, fa sì che il file resti aperto anche fra diverse chiamate a `getservbyname` e `getservbyport`; di default dopo una chiamata a queste funzioni il file viene chiuso, cosicché una successiva chiamata a `getservent` riparte dall'inizio. La terza funzione, `endservent`, provvede semplicemente a chiudere il file.

Queste tre funzioni per la lettura sequenziale di nuovo sono presenti per ciascuno dei vari tipi di informazione relative alle reti di tab. 16.7; questo significa che esistono altrettante funzioni nella forma `setXXXent`, `getXXXent` e `endXXXent`, analoghe alle precedenti per la risoluzione dei servizi, che abbiamo riportato in tab. 16.8. Essendo, a parte il tipo di informazione che viene trattato, sostanzialmente identiche nel funzionamento e di scarso utilizzo, non staremo a trattarle una per una, rimandando alle rispettive pagine di manuale.

16.1.4 Le funzioni avanzate per la risoluzione dei nomi

Quelle illustrate nella sezione precedente sono le funzioni classiche per la risoluzione di nomi ed indirizzi IP, ma abbiamo già visto come esse soffrano di vari inconvenienti come il fatto che usano informazioni statiche, e non prevedono la possibilità di avere diverse classi di indirizzi. Anche se sono state create delle estensioni o metodi diversi che permettono di risolvere alcuni di questi inconvenienti, comunque esse non forniscono una interfaccia sufficientemente generica.¹¹

Inoltre in genere quando si ha a che fare con i socket non esiste soltanto il problema della risoluzione del nome che identifica la macchina, ma anche quello del servizio a cui ci si vuole rivolgere. Per questo motivo con lo standard POSIX 1003.1-2001 sono state indicate come deprecate le varie funzioni `gethostbyaddr`, `gethostbyname`, `getipnodebyname` e `getipnodebyaddr` ed è stata introdotta una interfaccia completamente nuova.

¹¹rimane ad esempio il problema generico che si deve sapere in anticipo quale tipo di indirizzi IP (IPv4 o IPv6) corrispondono ad un certo nome a dominio.

La prima funzione di questa interfaccia è `getaddrinfo`, che combina le funzionalità delle precedenti `getipnodebyname`, `getipnodebyaddr`, `getservbyname` e `getservbyport`, consentendo di ottenere contemporaneamente sia la risoluzione di un indirizzo simbolico che del nome di un servizio; la funzione è stata introdotta, insieme a `getnameinfo` che vedremo più avanti, nell’RFC 2553 ed il suo prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints,
                struct addrinfo **res)
    Esegue una risoluzione di un nome a dominio e di un nome di servizio.
```

La funzione ritorna 0 in caso di successo e un codice di errore diverso da zero per un errore.

La funzione prende come primo argomento il nome della macchina che si vuole risolvere, specificato tramite la stringa `node`. Questo argomento, oltre ad un comune nome a dominio, può indicare anche un indirizzo numerico in forma *dotted-decimal* per IPv4 o in formato esadecimale per IPv6. Si può anche specificare il nome di una rete invece che di una singola macchina. Il secondo argomento, `service`, specifica invece il nome del servizio che si intende risolvere. Per uno dei due argomenti si può anche usare il valore `NULL`, nel qual caso la risoluzione verrà effettuata soltanto sulla base del valore dell’altro.

Il terzo argomento, `hints`, deve essere invece un puntatore ad una struttura `addrinfo` usata per dare dei *suggerimenti* al procedimento di risoluzione riguardo al protocollo o del tipo di socket che si intenderà utilizzare; `getaddrinfo` infatti permette di effettuare ricerche generiche sugli indirizzi, usando sia IPv4 che IPv6, e richiedere risoluzioni sui nomi dei servizi indipendentemente dal protocollo (ad esempio TCP o UDP) che questi possono utilizzare.

Come ultimo argomento in `res` deve essere passato un puntatore ad una variabile (di tipo puntatore ad una struttura `addrinfo`) che verrà utilizzata dalla funzione per riportare (come *value result argument*) i propri risultati. La funzione infatti è rientrante, ed alloca autonomamente tutta la memoria necessaria in cui verranno riportati i risultati della risoluzione. La funzione scriverà all’indirizzo puntato da `res` il puntatore iniziale ad una *linked list* di strutture di tipo `addrinfo` contenenti tutte le informazioni ottenute.

```
struct addrinfo
{
    int ai_flags;                /* Input flags. */
    int ai_family;              /* Protocol family for socket. */
    int ai_socktype;            /* Socket type. */
    int ai_protocol;           /* Protocol for socket. */
    socklen_t ai_addrlen;       /* Length of socket address. */
    struct sockaddr *ai_addr;   /* Socket address for socket. */
    char *ai_canonname;         /* Canonical name for service location. */
    struct addrinfo *ai_next;   /* Pointer to next in list. */
};
```

Figura 16.5: La struttura `addrinfo` usata nella nuova interfaccia POSIX per la risoluzione di nomi a dominio e servizi.

Come illustrato la struttura `addrinfo`, la cui definizione è riportata in fig. 16.5, viene usata sia in ingresso, per passare dei valori di controllo alla funzione, che in uscita, per ricevere i risultati. La definizione è ripresa direttamente dal file `netdb.h` in cui questa struttura viene dichiarata, la pagina di manuale riporta `size_t` come tipo di dato per il campo `ai_addrlen`,

qui viene usata quanto previsto dallo standard POSIX, in cui viene utilizzato `socklen_t`; i due tipi di dati sono comunque equivalenti.

Il primo campo, `ai_flags`, è una maschera binaria di bit che permettono di controllare le varie modalità di risoluzione degli indirizzi, che viene usato soltanto in ingresso. I tre campi successivi `ai_family`, `ai_socktype`, e `ai_protocol` contengono rispettivamente la famiglia di indirizzi, il tipo di socket e il protocollo, in ingresso vengono usati per impostare una selezione (impostandone il valore nella struttura puntata da `hints`), mentre in uscita indicano il tipo di risultato contenuto nella struttura.

Tutti i campi seguenti vengono usati soltanto in uscita e devono essere nulli o `NULL` in ingresso; il campo `ai_addrlen` indica la dimensione della struttura degli indirizzi ottenuta come risultato, il cui contenuto sarà memorizzato nella struttura `sockaddr` posta all'indirizzo puntato dal campo `ai_addr`. Il campo `ai_canonname` è un puntatore alla stringa contenente il nome canonico della macchina, ed infine, quando la funzione restituisce più di un risultato, `ai_next` è un puntatore alla successiva struttura `addrinfo` della lista.

Ovviamente non è necessario dare dei suggerimenti in ingresso, ed usando `NULL` come valore per l'argomento `hints` si possono compiere ricerche generiche. Se però si specifica un valore non nullo questo deve puntare ad una struttura `addrinfo` precedentemente allocata nella quale siano stati opportunamente impostati i valori dei campi `ai_family`, `ai_socktype`, `ai_protocol` ed `ai_flags`.

I due campi `ai_family` e `ai_socktype` prendono gli stessi valori degli analoghi argomenti della funzione `socket`; in particolare per `ai_family` si possono usare i valori di tab. 14.1 ma sono presi in considerazione solo `AF_INET` e `AF_INET6`, mentre se non si vuole specificare nessuna famiglia di indirizzi si può usare il valore `AF_UNSPEC`. Allo stesso modo per `ai_socktype` si possono usare i valori illustrati in sez. 14.1.4 per indicare per quale tipo di socket si vuole risolvere il servizio indicato, anche se i soli significativi sono `SOCK_STREAM` e `SOCK_DGRAM`; in questo caso, se non si vuole effettuare nessuna risoluzione specifica, si potrà usare un valore nullo.

Il campo `ai_protocol` permette invece di effettuare la selezione dei risultati per il nome del servizio usando il numero identificativo del rispettivo protocollo di trasporto (i cui valori possibili sono riportati in `/etc/protocols`); di nuovo i due soli valori utilizzabili sono quelli relativi a UDP e TCP, o il valore nullo che indica di ignorare questo campo nella selezione.

Infine gli ultimi dettagli si controllano con il campo `ai_flags`; che deve essere impostato come una maschera binaria; i bit di questa variabile infatti vengono usati per dare delle indicazioni sul tipo di risoluzione voluta, ed hanno valori analoghi a quelli visti in sez. 16.1.3 per `getipnodebyname`; il valore di `ai_flags` può essere impostata con un OR aritmetico delle costanti di tab. 16.9, ciascuna delle quali identifica un bit della maschera.

Nella seconda parte della tabella si sono riportati i valori delle costanti aggiunte a partire dalla *glibc* 2.3.4 per gestire la internazionalizzazione dei nomi a dominio (IDN o *Internationalized Domain Names*) secondo quanto specificato nell'RFC 3490 (potendo cioè usare codifiche di caratteri che consentono l'espressione di nomi a dominio in qualunque lingua).

Come accennato passando un valore `NULL` per l'argomento `hints` si effettua una risoluzione generica, equivalente ad aver impostato un valore nullo per `ai_family` e `ai_socktype`, un valore `AF_UNSPEC` per `ai_family` e il valore `(AI_V4MAPPED|AI_ADDRCONFIG)` per `ai_flags`.

La funzione restituisce un valore nullo in caso di successo, o un codice in caso di errore. I valori usati come codice di errore sono riportati in tab. 16.10; dato che la funzione utilizza altre funzioni e chiamate al sistema per ottenere il suo risultato in generale il valore di `errno` non è significativo, eccetto il caso in cui si sia ricevuto un errore di `EAI_SYSTEM`, nel qual caso l'errore corrispondente è riportato tramite `errno`.

Costante	Significato
AI_ADDRCONFIG	Stesso significato dell'analogo di tab. 16.6.
AI_ALL	Stesso significato dell'analogo di tab. 16.6.
AI_CANONNAME	Richiede la restituzione del nome canonico della macchina, che verrà salvato in una stringa il cui indirizzo sarà restituito nel campo <code>ai_canonname</code> della prima struttura <code>addrinfo</code> dei risultati. Se il nome canonico non è disponibile al suo posto viene restituita una copia di <code>node</code> .
AI_NUMERICHOST	Se impostato il nome della macchina specificato con <code>node</code> deve essere espresso in forma numerica, altrimenti sarà restituito un errore <code>EAI_NONAME</code> (vedi tab. 16.10), in questo modo si evita ogni chiamata alle funzioni di risoluzione.
AI_NUMERICSERVICE	Analogo di <code>AI_NUMERICHOST</code> per la risoluzione di un servizio, con <code>service</code> che deve essere espresso in forma numerica.
AI_PASSIVE	Viene utilizzato per ottenere un indirizzo in formato adatto per una successiva chiamata a <code>bind</code> . Se specificato quando si è usato <code>NULL</code> come valore per <code>node</code> gli indirizzi restituiti saranno inizializzati al valore generico (<code>INADDR_ANY</code> per IPv4 e <code>IN6ADDR_ANY_INIT</code> per IPv6), altrimenti verrà usato l'indirizzo dell'interfaccia di <i>loopback</i> . Se invece non è impostato gli indirizzi verranno restituiti in formato adatto ad una chiamata a <code>connect</code> o <code>sendto</code> .
AI_V4MAPPED	Stesso significato dell'analogo di tab. 16.6.
AI_CANONIDN	Se il nome canonico richiesto con <code>AI_CANONNAME</code> è codificato con questo flag la codifica viene convertita in forma leggibile nella localizzazione corrente.
AI_IDN	Se specificato il nome viene convertito, se necessario, nella codifica IDN, usando la localizzazione corrente.
AI_IDN_ALLOW_UNASSIGNED	attiva il controllo <code>IDNA_ALLOW_UNASSIGNED</code> .
AI_AI_IDN_USE_STD3_ASCII_RULES	attiva il controllo <code>IDNA_USE_STD3_ASCII_RULES</code>

Tabella 16.9: Costanti associate ai bit del campo `ai_flags` della struttura `addrinfo`.

Costante	Significato
EAI_ADDRFAMILY	La richiesta non ha nessun indirizzo di rete per la famiglia di indirizzi specificata.
EAI_AGAIN	Il DNS ha restituito un errore di risoluzione temporaneo, si può ritentare in seguito.
EAI_BADFLAGS	Il campo <code>ai_flags</code> contiene dei valori non validi per i flag o si è richiesto <code>AI_CANONNAME</code> con <code>name</code> nullo.
EAI_FAIL	Il DNS ha restituito un errore di risoluzione permanente.
EAI_FAMILY	La famiglia di indirizzi richiesta non è supportata.
EAI_MEMORY	È stato impossibile allocare la memoria necessaria alle operazioni.
EAI_NODATA	La macchina specificata esiste, ma non ha nessun indirizzo di rete definito.
EAI_NONAME	Il nome a dominio o il servizio non sono noti, viene usato questo errore anche quando si specifica il valore <code>NULL</code> per entrambi gli argomenti <code>node</code> e <code>service</code> .
EAI_SERVICE	Il servizio richiesto non è disponibile per il tipo di socket richiesto, anche se può esistere per altri tipi di socket.
EAI_SOCKTYPE	Il tipo di socket richiesto non è supportato.
EAI_SYSTEM	C'è stato un errore di sistema, si può controllare <code>errno</code> per i dettagli.

Tabella 16.10: Costanti associate ai valori dei codici di errore della funzione `getaddrinfo`.

Come per i codici di errore di `gethostbyname` anche in questo caso è fornita una apposita funzione, simile a `strerror`, che consente di utilizzare direttamente il codice restituito dalla funzione per stampare a video un messaggio esplicativo; la funzione è `gai_strerror` ed il suo prototipo è:

```
#include <netdb.h>
const char *gai_strerror(int errcode)
```

Fornisce il messaggio corrispondente ad un errore di `getaddrinfo`.

La funzione ritorna il puntatore alla stringa contenente il messaggio di errore.

La funzione restituisce un puntatore alla stringa contenente il messaggio corrispondente dal codice di errore `errcode` ottenuto come valore di ritorno di `getaddrinfo`. La stringa è allocata staticamente, ma essendo costante ed accessibile in sola lettura, la funzione è rientrante.

Dato che ad un certo nome a dominio possono corrispondere più indirizzi IP (sia IPv4 che IPv6), e che un certo servizio può essere fornito su protocolli e tipi di socket diversi, in generale, a meno di non aver eseguito una selezione specifica attraverso l'uso di `hints`, si otterrà una diversa struttura `addrinfo` per ciascuna possibilità.

Ad esempio se si richiede la risoluzione del servizio *echo* per l'indirizzo `www.truelite.it`, e si imposta `AI_CANONNAME` per avere anche la risoluzione del nome canonico, si avrà come risposta della funzione la lista illustrata in fig. 16.6.

Figura 16.6: La *linked list* delle strutture `addrinfo` restituite da `getaddrinfo`.

Come primo esempio di uso di `getaddrinfo` vediamo un programma elementare di interrogazione del *resolver* basato questa funzione, il cui corpo principale è riportato in fig. 16.7. Il codice completo del programma, compresa la gestione delle opzioni in cui è gestita l'eventuale inizializzazione dell'argomento `hints` per restringere le ricerche su protocolli, tipi di socket o famiglie di indirizzi, è disponibile nel file `mygetaddr.c` dei sorgenti allegati alla guida.

Il corpo principale inizia controllando (1-5) il numero di argomenti passati, che devono essere sempre due, e corrispondere rispettivamente all'indirizzo ed al nome del servizio da risolvere. A questo segue la chiamata (7) alla funzione `getaddrinfo`, ed il successivo controllo (8-11) del suo corretto funzionamento, senza il quale si esce immediatamente stampando il relativo codice di errore.

Se la funzione ha restituito un valore nullo il programma prosegue inizializzando (12) il puntatore `ptr` che sarà usato nel successivo ciclo (14-35) di scansione della lista delle strutture `addrinfo` restituite dalla funzione. Prima di eseguire questa scansione (12) viene stampato il valore del nome canonico che è presente solo nella prima struttura.

La scansione viene ripetuta (14) fintanto che si ha un puntatore valido. La selezione principale è fatta sul campo `ai_family`, che stabilisce a quale famiglia di indirizzi fa riferimento la struttura in esame. Le possibilità sono due, un indirizzo IPv4 o IPv6, se nessuna delle due si verifica si provvede (27-30) a stampare un messaggio di errore ed uscire (questa eventualità non dovrebbe comunque mai verificarsi, almeno fintanto che la funzione `getaddrinfo` lavora correttamente).

Per ciascuno delle due possibili famiglie di indirizzi si estraggono le informazioni che poi verranno stampate alla fine del ciclo (31-34). Il primo caso esaminato (15-21) è quello degli indirizzi IPv4, nel qual caso prima se ne stampa l'identificazione (16) poi si provvede a ricavare la struttura degli indirizzi (17) indirizzata dal campo `ai_addr`, eseguendo un opportuno casting del puntatore per poter estrarre da questa la porta (18) e poi l'indirizzo (19) che verrà convertito con una chiamata ad `inet_ntop`.

La stessa operazione (21-27) viene ripetuta per gli indirizzi IPv6, usando la rispettiva struttura degli indirizzi. Si noti anche come in entrambi i casi per la chiamata a `inet_ntop`

```

1  /* remaining argument check */
2  if ((argc - optind) != 2) {
3      printf("Wrong_number_of_arguments_%d\n", argc - optind);
4      usage();
5  }
6  /* main body */
7  ret = getaddrinfo(argv[optind], argv[optind+1], &hint, &res);
8  if (ret != 0) {
9      printf("Resolution_error_%s\n", gai_strerror(ret));
10     exit(1);
11 }
12 ptr = res;
13 printf("Canonical_name_%s\n", ptr->ai_canonname); /* print cname */
14 while (ptr != NULL) { /* loop on list */
15     if (ptr->ai_family == AF_INET) { /* if IPv4 */
16         printf("IPv4_address:_\n");
17         addr = (struct sockaddr_in *) ptr->ai_addr; /* address */
18         port = ntohs(addr->sin_port); /* port */
19         string = inet_ntop(addr->sin_family, &addr->sin_addr,
20                             buffer, sizeof(buffer));
21     } else if (ptr->ai_family == AF_INET6) { /* if IPv6 */
22         printf("IPv6_address:_\n");
23         addr6 = (struct sockaddr_in6 *) ptr->ai_addr; /* address */
24         port = ntohs(addr6->sin6_port); /* port */
25         string = inet_ntop(addr6->sin6_family, &addr6->sin6_addr,
26                             buffer, sizeof(buffer));
27     } else { /* else is an error */
28         printf("Address_family_error\n");
29         exit(1);
30     }
31     printf("\tIndirizzo_%s\n", string);
32     printf("\tProtocollo_%i\n", ptr->ai_protocol);
33     printf("\tPorta_%i\n", port);
34     ptr = ptr->ai_next;
35 }
36 exit(0);

```

Figura 16.7: Esempio di codice per la risoluzione di un indirizzo.

si sia dovuto passare il puntatore al campo contenente l'indirizzo IP nella struttura puntata dal campo `ai_addr`.¹²

Una volta estratte dalla struttura `addrinfo` tutte le informazioni relative alla risoluzione richiesta e stampati i relativi valori, l'ultimo passo (34) è di estrarre da `ai_next` l'indirizzo della eventuale successiva struttura presente nella lista e ripetere il ciclo, fin tanto che, completata la scansione, questo avrà un valore nullo e si potrà terminare (36) il programma.

Si tenga presente che `getaddrinfo` non garantisce nessun particolare ordinamento della lista delle strutture `addrinfo` restituite, anche se usualmente i vari indirizzi IP (se ne è presente più di uno) sono forniti nello stesso ordine in cui vengono inviati dal server DNS. In particolare nulla garantisce che vengano forniti prima i dati relativi ai servizi di un determinato protocollo

¹²il meccanismo è complesso a causa del fatto che al contrario di IPv4, in cui l'indirizzo IP può essere espresso con un semplice numero intero, in IPv6 questo deve essere necessariamente fornito come struttura, e pertanto anche se nella struttura puntata da `ai_addr` sono presenti direttamente i valori finali, per l'uso con `inet_ntop` occorre comunque passare un puntatore agli stessi (ed il costruito `&addr6->sin6_addr` è corretto in quanto l'operatore `->` ha in questo caso precedenza su `&`).

o tipo di socket, se ne sono presenti di diversi. Se allora utilizziamo il nostro programma potremo verificare il risultato:

```
[piccardi@gont sources]$ ./mygetaddr -c gapil.truelite.it echo
Canonical name sources2.truelite.it
IPv4 address:
    Indirizzo 62.48.34.25
    Protocollo 6
    Porta 7
IPv4 address:
    Indirizzo 62.48.34.25
    Protocollo 17
    Porta 7
```

Una volta estratti i risultati dalla *linked list* puntata da `res` se questa non viene più utilizzata si dovrà avere cura di disallocare opportunamente tutta la memoria, per questo viene fornita l'apposita funzione `freeaddrinfo`, il cui prototipo è:

```
#include <netdb.h>
void freeaddrinfo(struct addrinfo *res)
    Libera la memoria allocata da una precedente chiamata a getaddrinfo.
```

La funzione non restituisce nessun codice di errore.

La funzione prende come unico argomento il puntatore `res`, ottenuto da una precedente chiamata a `getaddrinfo`, e scandisce la lista delle strutture per liberare tutta la memoria allocata. Dato che la funzione non ha valori di ritorno deve essere posta molta cura nel passare un valore valido per `res` ed usare un indirizzo non valido o già liberato può avere conseguenze non prevedibili.

Si tenga presente infine che se si copiano i risultati da una delle strutture `addrinfo` restituite nella lista indicizzata da `res`, occorre avere cura di eseguire una *deep copy* in cui si copiano anche tutti i dati presenti agli indirizzi contenuti nella struttura `addrinfo`, perché una volta disallocati i dati con `freeaddrinfo` questi non sarebbero più disponibili.

Anche la nuova interfaccia definita da POSIX prevede una nuova funzione per eseguire la risoluzione inversa e determinare nomi di servizi e di dominio dati i rispettivi valori numerici. La funzione che sostituisce le varie `gethostbyname`, `getipnodebyname` e `getservbyname` è `getnameinfo`, ed il suo prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
    char *host, size_t hostlen, char *serv, size_t servlen, int flags)
    Effettua una risoluzione di un indirizzo di rete in maniera indipendente dal protocollo.
```

La funzione ritorna 0 in caso di successo e un codice di errore diverso da zero per un errore.

La principale caratteristica di `getnameinfo` è che la funzione è in grado di eseguire una risoluzione inversa in maniera indipendente dal protocollo; il suo primo argomento `sa` infatti è il puntatore ad una struttura degli indirizzi generica, che può contenere sia indirizzi IPv4 che IPv6, la cui dimensione deve comunque essere specificata con l'argomento `salen`.

I risultati della funzione saranno restituiti nelle due stringhe puntate da `host` e `serv`, che dovranno essere state precedentemente allocate per una lunghezza massima che deve essere specificata con gli altri due argomenti `hostlen` e `servlen`. Quando non si è interessati ad uno dei due, si può passare il valore `NULL` come argomento, così che la corrispondente informazione non venga richiesta. Infine l'ultimo argomento `flags` è una maschera binaria i cui bit consentono di impostare le modalità con cui viene eseguita la ricerca, e deve essere specificato

attraverso l'OR aritmetico dei valori illustrati in tab. 16.11, nella seconda parte della tabella si sono aggiunti i valori introdotti con la *glibc* 2.3.4 per gestire la internazionalizzazione dei nomi a dominio.

Costante	Significato
NI_DGRAM	Richiede che venga restituito il nome del servizio su UDP invece che quello su TCP per quei pochi servizi (porte 512-214) che sono diversi nei due protocolli.
NI_NOFQDN	Richiede che venga restituita solo il nome della macchina all'interno del dominio al posto del nome completo (FQDN).
NI_NAMEREQD	Richiede la restituzione di un errore se il nome non può essere risolto.
NI_NUMERICHOST	Richiede che venga restituita la forma numerica dell'indirizzo (questo succede sempre se il nome non può essere ottenuto).
NI_NUMERICSERV	Richiede che il servizio venga restituito in forma numerica (attraverso il numero di porta).
NI_IDN	Se specificato il nome restituito viene convertito usando la localizzazione corrente, se necessario, nella codifica IDN.
NI_IDN_ALLOW_UNASSIGNED	attiva il controllo IDNA_ALLOW_UNASSIGNED.
NI_AI_IDN_USE_STD3_ASCII_RULES	attiva il controllo IDNA_USE_STD3_ASCII_RULES

Tabella 16.11: Costanti associate ai bit dell'argomento flags della funzione `getnameinfo`.

La funzione ritorna zero in caso di successo, e scrive i propri risultati agli indirizzi indicati dagli argomenti `host` e `serv` come stringhe terminate dal carattere NUL, a meno che queste non debbano essere troncate qualora la loro dimensione ecceda quelle specificate dagli argomenti `hostlen` e `servlen`. Sono comunque definite le due costanti `NI_MAXHOST` e `NI_MAXSERV`¹³ che possono essere utilizzate come limiti massimi. In caso di errore viene restituito invece un codice che assume gli stessi valori illustrati in tab. 16.10.

A questo punto possiamo fornire degli esempi di utilizzo della nuova interfaccia, adottandola per le precedenti implementazioni del client e del server per il servizio *echo*; dato che l'uso delle funzioni appena illustrate (in particolare di `getaddrinfo`) è piuttosto complesso, essendo necessaria anche una impostazione diretta dei campi dell'argomento `hints`, provveremo una interfaccia semplificata per i due casi visti finora, quello in cui si specifica nel client un indirizzo remoto per la connessione al server, e quello in cui si specifica nel server un indirizzo locale su cui porsi in ascolto.

La prima funzione della nostra interfaccia semplificata è `sockconn` che permette di ottenere un socket, connesso all'indirizzo ed al servizio specificati. Il corpo della funzione è riportato in fig. 16.8, il codice completo è nel file `SockUtil.c` dei sorgenti allegati alla guida, che contiene varie funzioni di utilità per l'uso dei socket.

La funzione prende quattro argomenti, i primi due sono le stringhe che indicano il nome della macchina a cui collegarsi ed il relativo servizio su cui sarà effettuata la risoluzione; seguono il protocollo da usare (da specificare con il valore numerico di `/etc/protocols`) ed il tipo di socket (al solito specificato con i valori illustrati in sez. 14.1.4). La funzione ritorna il valore del file descriptor associato al socket (un numero positivo) in caso di successo, o `-1` in caso di errore.

Per risolvere il problema di non poter passare indietro i valori di ritorno di `getaddrinfo` contenenti i relativi codici di errore si sono stampati i messaggi d'errore direttamente nella funzione; infatti non si può avere nessuna certezza che detti valori siano negativi e per cui stampare subito l'errore diventa necessario per evitare ogni possibile ambiguità nei confronti del valore di ritorno in caso di successo.

¹³in Linux le due costanti sono definite in `netdb.h` ed hanno rispettivamente il valore 1024 e 12.

```

1 int sockconn(char *host, char *serv, int prot, int type)
2 {
3     struct addrinfo hint, *addr, *save;
4     int res;
5     int sock;
6     memset(&hint, 0, sizeof(struct addrinfo));
7     hint.ai_family = PF_UNSPEC;           /* generic address (IPv4 or IPv6) */
8     hint.ai_protocol = prot;             /* protocol */
9     hint.ai_socktype = type;             /* socket type */
10    res = getaddrinfo(host, serv, &hint, &addr); /* calling getaddrinfo */
11    if (res != 0) {                        /* on error exit */
12        fprintf(stderr, "sockconn:_resolution_failed:");
13        fprintf(stderr, "_%s\n", gai_strerror(res));
14        errno = 0;                        /* clear errno */
15        return -1;
16    }
17    save = addr;
18    while (addr != NULL) {                /* loop on possible addresses */
19        sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
20        if (sock < 0) {                    /* on error */
21            if (addr->ai_next != NULL) {    /* if other addresses */
22                addr=addr->ai_next;        /* take next */
23                continue;                  /* restart cycle */
24            } else {                       /* else stop */
25                perror("sockconn:_cannot_create_socket");
26                return sock;
27            }
28        }
29        if ( (res = connect(sock, addr->ai_addr, addr->ai_addrlen) < 0) ) {
30            if (addr->ai_next != NULL) {    /* if other addresses */
31                addr=addr->ai_next;        /* take next */
32                close(sock);                /* close socket */
33                continue;                  /* restart cycle */
34            } else {                       /* else stop */
35                perror("sockconn:_cannot_connect");
36                close(sock);
37                return res;
38            }
39        } else break;                     /* ok, we are connected! */
40    }
41    freeaddrinfo(save);                    /* done, release memory */
42    return sock;
43 }

```

Figura 16.8: Il codice della funzione sockconn.

Una volta definite le variabili occorrenti (3-5) la funzione prima (6) azzerava il contenuto della struttura `hint` e poi provvede (7-9) ad inizializzarne i valori necessari per la chiamata (10) a `getaddrinfo`. Di quest'ultima si controlla (12-16) il codice di ritorno, in modo da stampare un avviso di errore, azzerare `errno` ed uscire in caso di errore.

Dato che ad una macchina possono corrispondere più indirizzi IP, e di tipo diverso (sia IPv4 che IPv6), mentre il servizio può essere in ascolto soltanto su uno solo di questi, si provvede a tentare la connessione per ciascun indirizzo restituito all'interno di un ciclo (18-40) di scansione della lista restituita da `getaddrinfo`, ma prima (17) si salva il valore del puntatore per poterlo riutilizzare alla fine per disallocare la lista.

Il ciclo viene ripetuto (18) fintanto che si hanno indirizzi validi, ed inizia (19) con l'apertura del socket; se questa fallisce si controlla (20) se sono disponibili altri indirizzi, nel qual caso si passa al successivo (21) e si riprende (22) il ciclo da capo; se non ve ne sono si stampa l'errore ritornando immediatamente (24-27).

Quando la creazione del socket ha avuto successo si procede (29) direttamente con la connessione, di nuovo in caso di fallimento viene ripetuto (30-38) il controllo se vi sono o no altri indirizzi da provare nella stessa modalità fatta in precedenza, aggiungendovi però in entrambi i casi (32 e (36) la chiusura del socket precedentemente aperto, che non è più utilizzabile.

Se la connessione ha avuto successo invece si termina (39) direttamente il ciclo, e prima di ritornare (31) il valore del file descriptor del socket si provvede (30) a liberare le strutture `addrinfo` allocate da `getaddrinfo` utilizzando il valore del relativo puntatore precedentemente (17) salvato. Si noti come per la funzione sia del tutto irrilevante se la struttura ritornata contiene indirizzi IPv6 o IPv4, in quanto si fa uso direttamente dei dati relativi alle strutture degli indirizzi di `addrinfo` che sono opachi rispetto all'uso della funzione `connect`.

Per usare questa funzione possiamo allora modificare ulteriormente il nostro programma client per il servizio *echo*; in questo caso rispetto al codice usato finora per collegarsi (vedi fig. 15.11) avremo una semplificazione per cui il corpo principale del nostro client diventerà quello illustrato in fig. 16.9, in cui le chiamate a `socket`, `inet_pton` e `connect` sono sostituite da una singola chiamata a `sockconn`. Inoltre il nuovo client (il cui codice completo è nel file `TCP_echo_fifth.c` dei sorgenti allegati) consente di utilizzare come argomento del programma un nome a dominio al posto dell'indirizzo numerico, e può utilizzare sia indirizzi IPv4 che IPv6.

```

1 int main(int argc, char *argv[])
2 {
3     /*
4      * Variables definition
5      */
6     int sock, i;
7     int reset = 0;
8     ...
9     /* call sockaddr to get a connected socket */
10    if ( (sock = sockconn(argv[optind], "echo", 6, SOCK_STREAM)) < 0) {
11        if (errno) perror("Socket_creation_error");
12        return 1;
13    }
14    ...
15    /* do read/write operations */
16    ClientEcho(stdin, sock);
17    /* normal exit */
18    return 0;
19 }

```

Figura 16.9: Il nuovo codice per la connessione del client *echo*.

La seconda funzione di ausilio che abbiamo creato è `sockbind`, il cui corpo principale è riportato in fig. 16.10 (al solito il sorgente completo è nel file `sockbind.c` dei sorgenti allegati alla guida). Come si può notare la funzione è del tutto analoga alla precedente `sockconn`, e prende gli stessi argomenti, però invece di eseguire una connessione con `connect` si limita a chiamare `bind` per collegare il socket ad una porta.

```

1 int sockbind(char *host, char *serv, int prot, int type)
2 {
3     struct addrinfo hint, *addr, *save;
4     int res;
5     int sock;
6     char buf[INET6_ADDRSTRLEN];
7     memset(&hint, 0, sizeof(struct addrinfo));
8     hint.ai_flags = AI_PASSIVE;           /* address for binding */
9     hint.ai_family = PF_UNSPEC;          /* generic address (IPv4 or IPv6) */
10    hint.ai_protocol = prot;              /* protocol */
11    hint.ai_socktype = type;              /* socket type */
12    res = getaddrinfo(host, serv, &hint, &addr); /* calling getaddrinfo */
13    if (res != 0) {                        /* on error exit */
14        fprintf(stderr, "sockbind:_resolution_failed:");
15        fprintf(stderr, "%s\n", gai_strerror(res));
16        errno = 0;                        /* clear errno */
17        return -1;
18    }
19    save = addr;                           /* saving for freeaddrinfo */
20    while (addr != NULL) {                  /* loop on possible addresses */
21        sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
22        if (sock < 0) {                     /* on error */
23            if (addr->ai_next != NULL) {    /* if other addresses */
24                addr=addr->ai_next;         /* take next */
25                continue;                  /* restart cycle */
26            } else {                       /* else stop */
27                perror("sockbind:_cannot_create_socket");
28                return sock;
29            }
30        }
31        if ( (res = bind(sock, addr->ai_addr, addr->ai_addrlen)) < 0) {
32            if (addr->ai_next != NULL) {    /* if other addresses */
33                addr=addr->ai_next;         /* take next */
34                close(sock);                /* close socket */
35                continue;                  /* restart cycle */
36            } else {                       /* else stop */
37                perror("sockbind:_cannot_connect");
38                close(sock);
39                return res;
40            }
41        } else break;                     /* ok, we are binded! */
42    }
43    freeaddrinfo(save);                    /* done, release memory */
44    return sock;
45 }

```

Figura 16.10: Il codice della funzione sockbind.

Dato che la funzione è pensata per essere utilizzata da un server ci si può chiedere a quale scopo mantenere l'argomento `host` quando l'indirizzo di questo è usualmente noto. Si ricordi però quanto detto in sez. 15.2.1, relativamente al significato della scelta di un indirizzo specifico come argomento di `bind`, che consente di porre il server in ascolto su uno solo dei possibili diversi indirizzi presenti su di una macchina. Se non si vuole che la funzione esegua `bind` su un indirizzo specifico, ma utilizzi l'indirizzo generico, occorrerà avere cura di passare un valore `NULL` come valore per l'argomento `host`; l'uso del valore `AI_PASSIVE` serve ad ottenere il valore generico nella rispettiva struttura degli indirizzi.

Come già detto la funzione è analoga a `sockconn` ed inizia azzerando ed inizializzando (6-11) opportunamente la struttura `hint` con i valori ricevuti come argomenti, soltanto che in questo caso si è usata (8) una impostazione specifica dei flag di `hint` usando `AI_PASSIVE` per indicare che il socket sarà usato per una apertura passiva. Per il resto la chiamata (12-18) a `getaddrinfo` e ed il ciclo principale (20-42) sono identici, solo che si è sostituita (31) la chiamata a `connect` con una chiamata a `bind`. Anche la conclusione (43-44) della funzione è identica.

Si noti come anche in questo caso si siano inserite le stampe degli errori sullo *standard error*, nonostante la funzione possa essere invocata da un demone. Nel nostro caso questo non è un problema in quanto se la funzione non ha successo il programma deve uscire immediatamente prima di essere posto in background, e può quindi scrivere gli errori direttamente sullo *standard error*.

```

1 int main(int argc, char *argv[])
2 {
3     /*
4      * Variables definition
5      */
6     int list_fd, conn_fd;
7     ...
8     /* Main code begin here */
9     if (compat) {
10         Signal(SIGCHLD, HandSigCHLD);          /* install signal handler */
11     } else {
12         SignalRestart(SIGCHLD, HandSigCHLD);    /* non restarting handler */
13     }
14     /* create and bind socket */
15     if ( (list_fd = sockbind(argv[optind], "echo", 6, SOCK_STREAM)) < 0) {
16         return 1;
17     }
18     ...
19 }

```

Figura 16.11: Nuovo codice per l'apertura passiva del server *echo*.

Con l'uso di questa funzione si può modificare anche il codice del nostro server *echo*, che rispetto a quanto illustrato nella versione iniziale di fig. 15.13 viene modificato nella forma riportata in fig. 16.11. In questo caso il socket su cui porsi in ascolto viene ottenuto (15-18) da `sockbind` che si cura anche della eventuale risoluzione di un indirizzo specifico sul quale si voglia far ascoltare il server.

16.2 Le opzioni dei socket

Benché dal punto di vista del loro uso come canali di trasmissione di dati i socket vengano trattati allo stesso modo dei file, acceduti tramite i file descriptor, e gestiti con le ordinarie funzioni di lettura e scrittura dei file, l'interfaccia standard usata per la gestione dei file generici non è comunque sufficiente a controllare la moltitudine di caratteristiche specifiche che li contraddistinguono, considerato tra l'altro che queste possono essere completamente diverse fra loro a seconda del tipo di socket e della relativa forma di comunicazione sottostante.

In questa sezione vedremo allora quali sono le funzioni dedicate alla gestione delle caratteristiche specifiche dei vari tipi di socket, che vengono raggruppate sotto il nome generico di

“*socket options*”, ma soprattutto analizzeremo quali sono queste opzioni e quali caratteristiche e comportamenti dei socket permettono di controllare.

16.2.1 Le funzioni di gestione delle opzioni dei socket

La modalità principale con cui si possono gestire le caratteristiche dei socket (ne vedremo delle ulteriori nelle prossime sezioni) è quella che passa attraverso l’uso di due funzioni di sistema generiche che permettono rispettivamente di impostarle e di recuperarne il valore corrente. La prima di queste due funzioni, quella usata per impostare le *socket options*, è `setsockopt`, ed il suo prototipo è:

```
#include <sys/socket.h>
#include <sys/types.h>
int setsockopt(int sock, int level, int optname, const void *optval, socklen_t optlen)
                                                    Imposta le opzioni di un socket.
```

La funzione ritorna 0 in caso di successo e `-1` per un errore, nel qual caso `errno` assumerà uno dei valori:

```
EBADF    il file descriptor sock non è valido.
EFAULT   l’indirizzo optval non è valido.
EINVAL   il valore di optlen non è valido.
ENOPROTOOPT l’opzione scelta non esiste per il livello indicato.
ENOTSOCK il file descriptor sock non corrisponde ad un socket.
```

Il primo argomento della funzione, `sock`, indica il socket su cui si intende operare; per indicare l’opzione da impostare si devono usare i due argomenti successivi, `level` e `optname`. Come abbiamo visto in sez. 13.2 i protocolli di rete sono strutturati su vari livelli, ed l’interfaccia dei socket può usarne più di uno. Si avranno allora funzionalità e caratteristiche diverse per ciascun protocollo usato da un socket, e quindi saranno anche diverse le opzioni che si potranno impostare per ciascun socket, a seconda del *livello* (trasporto, rete, ecc.) su cui si vuole andare ad operare.

Il valore di `level` seleziona allora il protocollo su cui vuole intervenire, mentre `optname` permette di scegliere su quale delle opzioni che sono definite per quel protocollo si vuole operare. In sostanza la selezione di una specifica opzione viene fatta attraverso una coppia di valori `level` e `optname` e chiaramente la funzione avrà successo soltanto se il protocollo in questione prevede quella opzione ed è utilizzato dal socket. Infine `level` prevede anche il valore speciale `SOL_SOCKET` usato per le opzioni generiche che sono disponibili per qualunque tipo di socket.

Livello	Significato
<code>SOL_SOCKET</code>	Opzioni generiche dei socket.
<code>SOL_IP</code>	Opzioni specifiche per i socket che usano IPv4.
<code>SOL_TCP</code>	Opzioni per i socket che usano TCP.
<code>SOL_IPV6</code>	Opzioni specifiche per i socket che usano IPv6.
<code>SOL_ICMPV6</code>	Opzioni specifiche per i socket che usano ICMPv6.

Tabella 16.12: Possibili valori dell’argomento `level` delle funzioni `setsockopt` e `getsockopt`.

I valori usati per `level`, corrispondenti ad un dato protocollo usato da un socket, sono quelli corrispondenti al valore numerico che identifica il suddetto protocollo in `/etc/protocols`; dato che la leggibilità di un programma non trarrebbe certo beneficio dall’uso diretto dei valori numerici, più comunemente si indica il protocollo tramite le apposite costanti `SOL_*` riportate in tab. 16.12, dove si sono riassunti i valori che possono essere usati per l’argomento `level`.

La notazione in questo caso è, purtroppo, abbastanza confusa: infatti in Linux il valore si può impostare sia usando le costanti `SOL_*`, che le analoghe `IPPROTO_*` (citare anche da Stevens in [?]); entrambe hanno gli stessi valori che sono equivalenti ai numeri di protocollo di `/etc/protocols`, con una eccezione specifica, che è quella del protocollo ICMP, per la quale non esiste una costante, il che è comprensibile dato che il suo valore, 1, è quello che viene assegnato a `SOL_SOCKET`.

Il quarto argomento, `optval` è un puntatore ad una zona di memoria che contiene i dati che specificano il valore dell'opzione che si vuole passare al socket, mentre l'ultimo argomento `optlen`,¹⁴ è la dimensione in byte dei dati presenti all'indirizzo indicato da `optval`. Dato che il tipo di dati varia a seconda dell'opzione scelta, occorrerà individuare qual è quello che deve essere usato, ed utilizzare le opportune variabili.

La gran parte delle opzioni utilizzano per `optval` un valore intero, se poi l'opzione esprime una condizione logica, il valore è sempre un intero, ma si dovrà usare un valore non nullo per abilitarla ed un valore nullo per disabilitarla. Se invece l'opzione non prevede di dover ricevere nessun tipo di valore si deve impostare `optval` a `NULL`. Un piccolo numero di opzioni però usano dei tipi di dati peculiari, è questo il motivo per cui `optval` è stato definito come puntatore generico.

La seconda funzione usata per controllare le proprietà dei socket è `getsockopt`, che serve a leggere i valori delle opzioni dei socket ed a farsi restituire i dati relativi al loro funzionamento; il suo prototipo è:

```
#include <sys/socket.h>
#include <sys/types.h>
int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen)
    Legge le opzioni di un socket.
```

La funzione ritorna 0 in caso di successo e -1 per un errore, nel qual caso `errno` assumerà uno dei valori:

`EBADF` il file descriptor `sock` non è valido.
`EFAULT` l'indirizzo `optval` o quello di `optlen` non è valido.
`ENOPROTOPT` l'opzione scelta non esiste per il livello indicato.
`ENOTSOCK` il file descriptor `sock` non corrisponde ad un socket.

I primi tre argomenti sono identici ed hanno lo stesso significato di quelli di `setsockopt`, anche se non è detto che tutte le opzioni siano definite per entrambe le funzioni. In questo caso `optval` viene usato per ricevere le informazioni ed indica l'indirizzo a cui andranno scritti i dati letti dal socket, infine `optlen` diventa un puntatore ad una variabile che viene usata come *value result argument* per indicare, prima della chiamata della funzione, la lunghezza del buffer allocato per `optval` e per ricevere indietro, dopo la chiamata della funzione, la dimensione effettiva dei dati scritti su di esso. Se la dimensione del buffer allocato per `optval` non è sufficiente si avrà un errore.

16.2.2 Le opzioni generiche

Come accennato esiste un insieme generico di opzioni dei socket che possono applicarsi a qualunque tipo di socket, indipendentemente da quale protocollo venga poi utilizzato. Una descrizione di queste opzioni è generalmente disponibile nella settima sezione delle pagine di manuale; nel caso specifico la si può consultare con `man 7 socket`. Se si vuole operare su

¹⁴questo argomento è in realtà sempre di tipo `int`, come era nelle `libc4` e `libc5`; l'uso di `socklen_t` è stato introdotto da POSIX (valgono le stesse considerazioni per l'uso di questo tipo di dato fatte in sez. 15.2.4) ed adottato dalla *glibc*.

queste opzioni generiche il livello da utilizzare è `SOL_SOCKET`; si è riportato un elenco di queste opzioni in tab. 16.13.

Opzione	get	set	flag	Tipo	Descrizione
<code>SO_ACCEPTCONN</code>	•			int	Indica se il socket è in ascolto.
<code>SO_ATTACH_FILTER</code>		•		sock_fprog	Aggancia un filtro BPF al socket.
<code>SO_BINDTODEVICE</code>	•	•		char *	Lega il socket ad un dispositivo.
<code>SO_BROADCAST</code>	•	•	•	int	Attiva o disattiva il <i>broadcast</i> .
<code>SO_BSDCOMPAT</code>	•	•	•	int	Abilita la compatibilità con BSD.
<code>SO_BUSY_POLL</code>	•	•	•	int	Attiva il “ <i>busy poll</i> ” sul socket.
<code>SO_DEBUG</code>	•	•	•	int	Abilita il debugging sul socket.
<code>SO_DETACH_FILTER</code>		•	•	int	Rimuove il filtro BPF agganciato al socket.
<code>SO_DOMAIN</code>	•			int	Legge il tipo di socket.
<code>SO_DONTROUTE</code>	•	•	•	int	Non invia attraverso un gateway.
<code>SO_ERROR</code>	•			int	Riceve e cancella gli errori pendenti.
<code>SO_KEEPAIVE</code>	•	•	•	int	Controlla l’attività della connessione.
<code>SO_LINGER</code>	•	•		linger	Indugia nella chiusura con dati da spedire.
<code>SO_LOCK_FILTER</code>		•	•	int	Blocca il filtro BPF agganciato al socket.
<code>SO_MARK</code>	•	•		int	Imposta un “ <i>firewall mark</i> ” sul socket.
<code>SO_OOBINLINE</code>	•	•	•	int	Lascia in linea i dati <i>out-of-band</i> .
<code>SO_PASSCRED</code>	•	•	•	int	Abilita la ricezione di credenziali.
<code>SO_PEEK_OFF</code>	•	•	•	int	Imposta il valore del “ <i>peek offset</i> ”.
<code>SO_PEERCREC</code>	•			ucred	Restituisce le credenziali del processo remoto.
<code>SO_PRIORITY</code>	•	•		int	Imposta la priorità del socket.
<code>SO_PROTOCOL</code>	•			int	Ottiene il protocollo usato dal socket.
<code>SO_RCVBUF</code>	•	•		int	Imposta dimensione del buffer di ricezione.
<code>SO_RCVBUFFORCE</code>	•	•		int	Forza dimensione del buffer di ricezione.
<code>SO_RCVLOWAT</code>	•	•	•	int	Basso livello sul buffer di ricezione.
<code>SO_RCVTIMEO</code>	•	•		timeval	Timeout in ricezione.
<code>SO_REUSEADDR</code>	•	•	•	int	Consente il riutilizzo di un indirizzo locale.
<code>SO_REUSEPORT</code>	•	•	•	int	Consente il riutilizzo di una porta.
<code>SO_RXQ_OVFL</code>	•	•	•	int	Richiede messaggio ancillare con pacchetti persi.
<code>SO_SNDBUF</code>	•	•		int	Imposta dimensione del buffer di trasmissione.
<code>SO_SNDBUFFORCE</code>	•	•		int	Forza dimensione del buffer di trasmissione.
<code>SO_SNDLOWAT</code>	•	•		int	Basso livello sul buffer di trasmissione.
<code>SO_SNDTIMEO</code>	•	•		timeval	Timeout in trasmissione.
<code>SO_TIMESTAMP</code>	•	•	•	int	Abilita/disabilita la ricezione dei <i>timestamp</i> .
<code>SO_TYPE</code>	•			int	Restituisce il tipo di socket.

Tabella 16.13: Le opzioni disponibili al livello `SOL_SOCKET`.

La tabella elenca le costanti che identificano le singole opzioni da usare come valore per `optname`; le due colonne seguenti indicano per quali delle due funzioni (`getsockopt` o `setsockopt`) l’opzione è disponibile, mentre la colonna successiva indica, quando si ha a che fare con un valore di `optval` intero, se l’opzione è da considerare un numero o un valore logico. Si è inoltre riportato sulla quinta colonna il tipo di dato usato per `optval` ed una breve descrizione del significato delle singole opzioni sulla sesta.

Le descrizioni delle opzioni presenti in tab. 16.13 sono estremamente sommarie, è perciò necessario fornire un po’ più di informazioni. Alcune opzioni inoltre hanno una notevole rilevanza nella gestione dei socket, e pertanto il loro utilizzo sarà approfondito separatamente in sez. 16.2.3. Quello che segue è quindi soltanto un elenco più dettagliato della breve descrizione di tab. 16.13 sul significato delle varie opzioni:

`SO_ACCEPTCONN`

questa opzione permette di rilevare se il socket su cui opera è stato posto in modalità di ricezione di eventuali connessioni con una chiamata a `listen`. L’opzione può essere usata soltanto con `getsockopt` ed utilizza per `optval` un intero in cui viene restituito 1 se il socket è in ascolto e 0 altrimenti.

SO_ATTACH_FILTER

questa opzione permette di agganciare ad un socket un filtro di selezione dei pacchetti con la stessa sintassi del BPF (*Berkley Packet Filter*) di BSD, che consente di selezionare, fra tutti quelli ricevuti, verranno letti. Può essere usata solo con `setsockopt` ed utilizza per `optval` un puntatore ad una struttura `sock_fprog` (definita in `linux/filter.h`). Questa opzione viene usata principalmente con i socket di tipo `AF_PACKET` (torneremo su questo in sez. 17.3.3) dalla libreria `libpcap` per implementare programmi di cattura dei pacchetti, e per questo tipo di applicazione è opportuno usare sempre quest'ultima.¹⁵

SO_BINDTODEVICE

questa opzione permette di *legare* il socket ad una particolare interfaccia, in modo che esso possa ricevere ed inviare pacchetti solo su quella. L'opzione richiede per `optval` il puntatore ad una stringa contenente il nome dell'interfaccia (ad esempio `eth0`); utilizzando una stringa nulla o un valore nullo per `optlen` si può rimuovere un precedente collegamento.

Il nome della interfaccia deve essere specificato con una stringa terminata da uno zero e di lunghezza massima pari a `IFNAMSIZ`; l'opzione è effettiva solo per alcuni tipi di socket, ed in particolare per quelli della famiglia `AF_INET`; non è invece supportata per i *packet socket* (vedi sez. 17.3.1).

SO_BROADCAST questa opzione abilita il *broadcast*: quando abilitata i socket di tipo `SOCK_DGRAM` riceveranno i pacchetti inviati all'indirizzo di *broadcast* e potranno scrivere pacchetti su tale indirizzo. Prende per `optval` un intero usato come valore logico. L'opzione non ha effetti su un socket di tipo `SOCK_STREAM`.

SO_BSDCOMPAT questa opzione abilita la compatibilità con il comportamento di BSD (in particolare ne riproduce alcuni bug). Attualmente è una opzione usata solo per il protocollo UDP e ne è prevista la rimozione. L'opzione utilizza per `optval` un intero usato come valore logico.

Quando viene abilitata gli errori riportati da messaggi ICMP per un socket UDP non vengono passati al programma in *user space*. Con le versioni 2.0.x del kernel erano anche abilitate altre opzioni di compatibilità per i socket raw (modifiche casuali agli header, perdita del flag di *broadcast*) che sono state rimosse con il passaggio al 2.2; è consigliato correggere i programmi piuttosto che usare questa funzione. Dal kernel 2.4 viene ignorata, e dal 2.6 genera un messaggio di log del kernel.

SO_BUSY_POLL questa opzione, presente dal kernel 3.11, imposta un tempo approssimato in microsecondi, per cui in caso di ricezione bloccante verrà eseguito un “*busy poll*”,¹⁶ da indicare in `optval` con un valore intero. Si tratta di una opzione utilizzabile solo con socket che ricevono dati da un dispositivo di rete che la supporti, e che consente di ridurre la latenza per alcune applicazioni, ma che comporta un maggiore utilizzo della CPU (e quindi di energia); per questo il valore può essere aumentato solo da processi con i privilegi di amministratore (in particolare con la *capability* `CAP_NET_ADMIN`). Il valore di default viene

¹⁵la trattazione del BPF va al di là dell'argomento di questa sezione per la documentazione si consulti il file `networking/filter.txt` nella documentazione del kernel.

¹⁶con *busy poll* si fa riferimento al *polling* su una risorsa occupata; si continuerà cioè a tentare di leggere anche quando non ci sono dati senza portare il processo stato di *sleep*, in alcuni casi, quando ci si aspetta che i dati arrivino a breve, questa tecnica può dare un miglioramento delle prestazioni.

controllato dal file `busy_read` per le funzioni di lettura mentre il file `busy_poll` controlla il “*busy poll*” per `select` e `poll`.

SO_DEBUG questa opzione abilita il debugging delle operazioni dei socket; l’opzione utilizza per `optval` un intero usato come valore logico, e può essere utilizzata solo da un processo con i privilegi di amministratore (in particolare con la *capability* `CAP_NET_ADMIN`). L’opzione necessita inoltre dell’opportuno supporto nel kernel;¹⁷ quando viene abilitata una serie di messaggi con le informazioni di debug vengono inviati direttamente al sistema del kernel log.¹⁸

SO_DETACH_FILTER consente di distaccare un filtro precedentemente aggiunto ad un socket con l’opzione `SO_ATTACH_FILTER`, in genere non viene usata direttamente in quanto i filtri BPF vengono automaticamente rimossi alla chiusura del socket, il suo utilizzo è pertanto limitato ai rari casi in cui si vuole rimuovere un precedente filtro per inserirne uno diverso. Come `SO_ATTACH_FILTER` può essere usato solo `setsockopt` e prende per `optval` un intero usato come valore logico.

SO_DOMAIN questa opzione, presente dal kernel 2.6.32, legge il “*dominio*” (la famiglia di indirizzi) del socket. . Funziona solo con `getsockopt`, ed utilizza per `optval` un intero in cui verrà restituito il valore numerico che lo identifica (ad esempio `AF_INET`).

SO_DONTROUTE questa opzione richiede che l’invio dei pacchetti del socket sia eseguito soltanto verso una destinazione direttamente connessa, impedendo l’uso di un *gateway* e saltando ogni processo relativo all’uso della tabella di routing del kernel. Prende per `optval` un intero usato come valore logico. È equivalente all’uso del flag `MSG_DONTROUTE` su una `send` (vedi sez. 18.1.1).

SO_ERROR questa opzione riceve un errore presente sul socket; può essere utilizzata soltanto con `getsockopt` e prende per `optval` un valore intero, nel quale viene restituito il codice di errore, e la condizione di errore sul socket viene cancellata. Viene usualmente utilizzata per ricevere il codice di errore, come accennato in sez. 15.6.1, quando si sta osservando il socket con una `select` che ritorna a causa dello stesso.

SO_KEEPALIVE questa opzione abilita un meccanismo di verifica della persistenza di una connessione associata al socket; è pertanto effettiva solo sui socket che supportano le connessioni, ed è usata principalmente con il TCP. L’opzione utilizza per `optval` un intero usato come valore logico. Maggiori dettagli sul suo funzionamento sono forniti in sez. 16.2.3.

SO_LINGER questa opzione controlla le modalità con cui viene chiuso un socket quando si utilizza un protocollo che supporta le connessioni (è pertanto usata con i socket TCP ed ignorata per UDP) e modifica il comportamento delle funzioni `close` e `shutdown`. L’opzione richiede che l’argomento `optval` sia una struttura

¹⁷deve cioè essere definita la macro di preprocessore `SOCK_DEBUGGING` nel file `include/net/sock.h` dei sorgenti del kernel, questo è sempre vero nei kernel delle serie superiori alla 2.3, per i kernel delle serie precedenti invece è necessario aggiungere a mano detta definizione; è inoltre possibile abilitare anche il tracciamento degli stati del TCP definendo la macro `STATE_TRACE` in `include/net/tcp.h`.

¹⁸si tenga presente che il comportamento è diverso da quanto avviene con BSD, dove l’opzione opera solo sui socket TCP, causando la scrittura di tutti i pacchetti inviati sulla rete su un buffer circolare che viene letto da un apposito programma, `trpt`.

di tipo `linger`, definita in `sys/socket.h` ed illustrata in fig. 16.15. Maggiori dettagli sul suo funzionamento sono forniti in sez. 16.2.3.

SO_LOCK_FILTER

consente di bloccare un filtro precedentemente aggiunto ad un socket con l'opzione `SO_ATTACH_FILTER`, in modo che non possa essere né rimosso né modificato, questo consente di impostare un filtro su un socket, bloccarlo e poi cedere i privilegi con la sicurezza che il filtro permarrà fino alla chiusura del socket. Come `SO_ATTACH_FILTER` può essere usato solo `setsockopt` e prende per `optval` un intero usato come valore logico.

SO_MARK

questa opzione, presente dal kernel 2.6.25, imposta un valore di marcatura sui pacchetti del socket. Questa è una funzionalità specifica di Linux, ottenibile anche con l'uso del target `MARK` del comando `iptables` (l'argomento è trattato in sez. 3.3.5 di [?]). Il valore di marcatura viene mantenuto all'interno dello stack di rete del kernel e può essere usato sia dal *netfilter*¹⁹ di Linux che per impostare politiche di routing avanzato. Il valore deve essere specificato in `optval` come un intero. L'opzione richiede i privilegi di amministratore con la *capability* `CAP_NET_ADMIN`.

SO_OOBINLINE

se questa opzione viene abilitata i dati *out-of-band* vengono inviati direttamente nel flusso di dati del socket (e sono quindi letti con una normale `read`) invece che restare disponibili solo per l'accesso con l'uso del flag `MSG_OOB` di `recvmsg`. L'argomento è trattato in dettaglio in sez. 18.1.4. L'opzione funziona soltanto con socket che supportino i dati *out-of-band* (non ha senso per socket UDP ad esempio), ed utilizza per `optval` un intero usato come valore logico.

SO_PASSCRED

questa opzione abilita sui socket unix-domain (vedi sez. 17.2) la ricezione dei messaggi di controllo di tipo `SCM_CREDENTIALS`. Prende come `optval` un intero usato come valore logico.

SO_PEEK_OFF

questa opzione, disponibile a partire dal kernel 3.4, imposta un "*peek offset*" sul socket (attualmente disponibile solo per i socket unix-domain (vedi sez. 17.2)). La funzione serve come ausilio per l'uso del flag `MSG_PEEK` di `recv` che consente di "sbirciare" nei dati di un socket, cioè di leggerli senza rimuoverli dalla coda in cui sono mantenuti, così che vengano restituiti anche da una successiva lettura ordinaria.

Un valore negativo (il default è `-1`) riporta alla situazione ordinaria in cui si "sbircia" a partire dalla testa della coda, un valore positivo consente di leggere a partire dalla posizione indicata nella coda e tutte le volte che si sbirciano dei dati il valore dell'offset viene automaticamente aumentato della quantità di dati sbirciati, in modo che si possa proseguire da dove si era arrivati. Il valore deve essere specificato in `optval` come intero.

SO_PEERCREC

questa opzione restituisce le credenziali del processo remoto connesso al socket; l'opzione è disponibile solo per socket unix-domain di tipo *stream* e anche per quelli di tipo *datagram* quando se ne crea una coppia con `socketpair`, e può essere usata solo con `getsockopt`. Utilizza per `optval` una apposita struttura `ucred` (vedi sez. 17.2).

¹⁹il *netfilter* è l'infrastruttura usata per il filtraggio dei pacchetti del kernel, per maggiori dettagli si consulti il cap. 2 di [?].

- SO_PRIORITY** questa opzione permette di impostare le priorità per tutti i pacchetti che sono inviati sul socket, prende per `optval` un valore intero. Con questa opzione il kernel usa il valore per ordinare le priorità sulle code di rete,²⁰ i pacchetti con priorità più alta vengono processati per primi, in modalità che dipendono dalla disciplina di gestione della coda. Nel caso di protocollo IP questa opzione permette anche di impostare i valori del campo *type of service* (noto come TOS, vedi sez. B.1.2) per i pacchetti uscenti. Per impostare una priorità al di fuori dell'intervallo di valori fra 0 e 6 sono richiesti i privilegi di amministratore con la *capability* `CAP_NET_ADMIN`.
- SO_PROTOCOL** questa opzione, presente dal kernel 2.6.32, legge il protocollo usato dal socket. Funziona solo con `getsockopt`, ed utilizza per `optval` un intero in cui verrà restituito il valore numerico che lo identifica (ad esempio `IPPROTO_TCP`).
- SO_RCVBUF** questa opzione imposta la dimensione del buffer di ricezione del socket. Prende per `optval` un intero indicante il numero di byte. Il valore di default ed il valore massimo che si può specificare come argomento per questa opzione sono impostabili tramite gli opportuni valori di `sysctl` (vedi sez. 16.4.1).
- Si tenga presente che nel caso di socket TCP il kernel alloca effettivamente una quantità di memoria doppia rispetto a quanto richiesto con `setsockopt` per entrambe le opzioni `SO_RCVBUF` e `SO_SNDBUF`. Questo comporta che una successiva lettura con `getsockopt` riporterà un valore diverso da quello impostato con `setsockopt`. Questo avviene perché TCP necessita dello spazio in più per mantenere dati amministrativi e strutture interne, e solo una parte viene usata come buffer per i dati, mentre il valore letto da `getsockopt` e quello riportato nei vari parametri di `sysctl`²¹ indica la memoria effettivamente impiegata. Si tenga presente inoltre che le modifiche alle dimensioni dei buffer di ricezione e trasmissione, per poter essere effettive, devono essere impostate prima della chiamata alle funzioni `listen` o `connect`.
- SO_RCVBUFFORCE** questa opzione, presente dal kernel 2.6.14, è identica a `SO_RCVBUF` ma consente ad un processo con i privilegi di amministratore (per la precisione con la *capability* `CAP_NET_ADMIN`) di impostare in valore maggiore del limite di `rmem_max`.
- SO_RCVLOWAT** questa opzione imposta il valore che indica il numero minimo di byte che devono essere presenti nel buffer di ricezione perché il kernel passi i dati all'utente, restituendoli ad una `read` o segnalando ad una `select` (vedi sez. 15.6.1) che ci sono dati in ingresso. L'opzione utilizza per `optval` un intero che specifica il numero di byte; con Linux questo valore è sempre 1 e può essere cambiato solo con i kernel a partire dal 2.4. Si tenga presente però che per i kernel prima del 2.6.28 sia `poll` che `select` non supportano questa funzionalità e ritornano comunque, indicando il socket come leggibile, non appena almeno un byte è presente, con una successiva lettura con `read` che si blocca fintanto che non diventa disponibile la quantità di byte indicati. Con `getsockopt` si può legge-

²⁰questo richiede che sia abilitato il sistema di *Quality of Service* disponibile con le opzioni di routing avanzato.

²¹cioè `wmem_max` e `rmem_max` in `/proc/sys/net/core` e `tcp_wmem` e `tcp_rmem` in `/proc/sys/net/ipv4`, vedi sez. 16.4.1.

re questo valore mentre `setsockopt` darà un errore di `ENOPROTOOPT` quando il cambiamento non è supportato.

SO_RCVTIMEO l'opzione permette di impostare un tempo massimo sulle operazioni di lettura da un socket, e prende per `optval` una struttura di tipo `timeval` (vedi fig. 4.15) identica a quella usata con `select`. Con `getsockopt` si può leggere il valore attuale, mentre con `setsockopt` si imposta il tempo voluto, usando un valore nullo per `timeval` il timeout viene rimosso.

Se l'opzione viene attivata tutte le volte che una delle funzioni di lettura (`read`, `readv`, `recv`, `recvfrom` e `recvmsg`) si blocca in attesa di dati per un tempo maggiore di quello impostato, essa ritornerà un valore -1 e la variabile `errno` sarà impostata con un errore di `EAGAIN` e `EWOULDBLOCK`, così come sarebbe avvenuto se si fosse aperto il socket in modalità non bloccante.²²

In genere questa opzione non è molto utilizzata se si ha a che fare con la lettura dei dati, in quanto è sempre possibile usare una `select` che consente di specificare un *timeout*; l'uso di `select` non consente però di impostare il timeout per l'uso di `connect`, per avere il quale si può ricorrere a questa opzione (nel qual caso il raggiungimento del *timeout* restituisce l'errore `EINPROGRESS`).

SO_REUSEADDR questa opzione permette di eseguire la funzione `bind` su indirizzi locali che siano già in uso da altri socket; l'opzione utilizza per `optval` un intero usato come valore logico. Questa opzione modifica il comportamento normale dell'interfaccia dei socket che fa fallire l'esecuzione della funzione `bind` con un errore di `EADDRINUSE` quando l'indirizzo locale (più propriamente il controllo viene eseguito sulla porta) è già in uso da parte di un altro socket. Maggiori dettagli sul suo funzionamento sono forniti in sez. 16.2.3.

SO_REUSEPORT questa opzione, presente a partire dal kernel 3.9, permette di far usare a più socket di tipo `AF_INET` o `AF_INET6` lo stesso indirizzo locale, e costituisce una estensione della precedente `SO_REUSEADDR`. Maggiori dettagli sul suo funzionamento sono forniti in sez. 16.2.3.

SO_RXQ_OVFL questa opzione, presente dal kernel 2.6.33, permette di abilitare o disabilitare sul socket la ricezione di un messaggio ancillare (tratteremo l'argomento in sez. 18.1.3) contenente un intero senza segno a 32 bit che indica il numero di pacchetti scartati sul socket fra l'ultimo pacchetto ricevuto e questo. L'opzione utilizza per `optval` un intero usato come valore logico.

SO_SNDLOWAT questa opzione imposta il valore che indica il numero minimo di byte che devono essere presenti nel buffer di trasmissione perché il kernel li invii al protocollo successivo, consentendo ad una `write` di ritornare o segnalando ad una `select` (vedi sez. 15.6.1) che è possibile eseguire una scrittura. L'opzione utilizza per `optval` un intero che specifica il numero di byte; con Linux questo valore è sempre 1 e non può essere cambiato; `getsockopt` leggerà questo valore mentre `setsockopt` darà un errore di `ENOPROTOOPT`.

SO_SNDBUF questa opzione imposta la dimensione del buffer di trasmissione del socket. Prende per `optval` un intero indicante il numero di byte. Il valore di default ed

²²in teoria, se il numero di byte presenti nel buffer di ricezione fosse inferiore a quello specificato da `SO_RCVLOWAT`, l'effetto potrebbe essere semplicemente quello di provocare l'uscita delle funzioni di lettura restituendo il numero di byte fino ad allora ricevuti.

il valore massimo che si possono specificare come argomento per questa opzione sono impostabili rispettivamente tramite gli opportuni valori di `sysctl` (vedi sez. 16.4.1).

SO_SNDBUFSIZE

questa opzione, presente dal kernel 2.6.14, è identica a `SO_SNDBUF` ma consente ad un processo con i privilegi di amministratore (per la precisione con la *capability* `CAP_NET_ADMIN`) di impostare in valore maggiore del limite di `wmem_max`.

SO_SNDTIMEO

l'opzione permette di impostare un tempo massimo sulle operazioni di scrittura su un socket, ed usa gli stessi valori di `SO_RCVTIMEO`. In questo caso però si avrà un errore di `EAGAIN` o `EWOULDBLOCK` per le funzioni di scrittura `write`, `writew`, `send`, `sendto` e `sendmsg` qualora queste restino bloccate per un tempo maggiore di quello specificato.

SO_TIMESTAMP

questa opzione, presente dal kernel 2.6.30, permette di abilitare o disabilitare sul socket la ricezione di un messaggio ancillare di tipo `SO_TIMESTAMP`. Il messaggio viene inviato con livello `SOL_SOCKET` ed nel campo `cmsg_data` (per i dettagli si veda sez. 18.1.3) viene impostata una struttura `timeval` che indica il tempo di ricezione dell'ultimo pacchetto passato all'utente. L'opzione utilizza per `optval` un intero usato come valore logico.

SO_TYPE

questa opzione permette di leggere il tipo di socket su cui si opera; funziona solo con `getsockopt`, ed utilizza per `optval` un intero in cui verrà restituito il valore numerico che lo identifica (ad esempio `SOCK_STREAM`).

16.2.3 L'uso delle principali opzioni dei socket

La descrizione sintetica del significato delle opzioni generiche dei socket, riportata nell'elenco in sez. 16.2.2, è necessariamente sintetica, alcune di queste però possono essere utilizzate per controllare delle funzionalità che hanno una notevole rilevanza nella programmazione dei socket. Per questo motivo faremo in questa sezione un approfondimento sul significato delle opzioni generiche più importanti.

L'opzione SO_KEEPALIVE

La prima opzione da approfondire è `SO_KEEPALIVE` che permette di tenere sotto controllo lo stato di una connessione. Una connessione infatti resta attiva anche quando non viene effettuato alcun traffico su di essa; è allora possibile, in caso di una interruzione completa della rete, che la caduta della connessione non venga rilevata, dato che sulla stessa non passa comunque alcun traffico.

Se si imposta questa opzione, è invece cura del kernel inviare degli appositi messaggi sulla rete, detti appunto *keep-alive*, per verificare se la connessione è attiva. L'opzione funziona soltanto con i socket che supportano le connessioni (non ha senso per socket UDP ad esempio) e si applica principalmente ai socket TCP.

Con le impostazioni di default (che sono riprese da BSD) Linux emette un messaggio di *keep-alive* (in sostanza un segmento ACK vuoto, cui sarà risposto con un altro segmento ACK vuoto) verso l'altro capo della connessione se questa è rimasta senza traffico per più di due ore. Se è tutto a posto il messaggio viene ricevuto e verrà emesso un segmento ACK di

risposta, alla cui ricezione ripartirà un altro ciclo di attesa per altre due ore di inattività; il tutto avviene all'interno del kernel e le applicazioni non riceveranno nessun dato.

Qualora ci siano dei problemi di rete si possono invece verificare i due casi di terminazione precoce del server già illustrati in sez. 15.5.3. Il primo è quello in cui la macchina remota ha avuto un crollo del sistema ed è stata riavviata, per cui dopo il riavvio la connessione non esiste più.²³ In questo caso all'invio del messaggio di *keep-alive* si otterrà come risposta un segmento RST che indica che l'altro capo non riconosce più l'esistenza della connessione ed il socket verrà chiuso riportando un errore di `ECONNRESET`.

Se invece non viene ricevuta nessuna risposta (indice che la macchina non è più raggiungibile) l'emissione dei messaggi viene ripetuta ad intervalli di 75 secondi per un massimo di 9 volte²⁴ (per un totale di 11 minuti e 15 secondi) dopo di che, se non si è ricevuta nessuna risposta, il socket viene chiuso dopo aver impostato un errore di `ETIMEDOUT`. Qualora la connessione si sia ristabilita e si riceva un successivo messaggio di risposta il ciclo riparte come se niente fosse avvenuto. Infine se si riceve come risposta un pacchetto ICMP di destinazione irraggiungibile (vedi sez. B.3), verrà restituito l'errore corrispondente.

In generale questa opzione serve per individuare una caduta della connessione anche quando non si sta facendo traffico su di essa. Viene usata principalmente sui server per evitare di mantenere impegnate le risorse che verrebbero dedicate a trattare delle connessioni che in realtà sono già terminate (quelle che vengono anche chiamate connessioni *semi-aperte*); in tutti quei casi cioè in cui il server si trova in attesa di dati in ingresso su una connessione che non arriveranno mai o perché il client sull'altro capo non è più attivo o perché non è più in grado di comunicare con il server via rete.

```

1 int main(int argc, char *argv[])
2 {
3     /*
4      * Variables definition
5      */
6     int list_fd, conn_fd;
7     int waiting = 0;
8     int keepalive = 0;
9     ...
10    ...
11
12    if (pid == 0) {          /* child */
13        close(list_fd);      /* close listening socket */
14        if (keepalive) {     /* enable keepalive ? */
15            setsockopt(conn_fd, SOL_SOCKET, SO_KEEPALIVE,
16                        &keepalive, sizeof(keepalive));
17        }
18        ServEcho(conn_fd);   /* handle echo */
19        ...
20 }

```

Figura 16.12: La sezione della nuova versione del server del servizio *echo* che prevede l'attivazione del *keepalive* sui socket.

²³si ricordi che un normale riavvio o il crollo dell'applicazione non ha questo effetto, in quanto in tal caso si passa sempre per la chiusura del processo, e questo, come illustrato in sez. 5.1.2, comporta anche la regolare chiusura del socket con l'invio di un segmento FIN all'altro capo della connessione.

²⁴entrambi questi valori possono essere modificati a livello di sistema (cioè per tutti i socket) con gli opportuni parametri illustrati in sez. 16.4.1 ed a livello di singolo socket con le opzioni `TCP_KEE*` di sez. 16.2.5.

Abilitandola dopo un certo tempo le connessioni effettivamente terminate verranno comunque chiuse per cui, utilizzando ad esempio una `select`, se ne potrà rilevare la conclusione e ricevere il relativo errore. Si tenga presente però che non può avere la certezza assoluta che un errore di `ETIMEDOUT` ottenuto dopo aver abilitato questa opzione corrisponda necessariamente ad una reale conclusione della connessione, il problema potrebbe anche essere dovuto ad un problema di routing che perduri per un tempo maggiore di quello impiegato nei vari tentativi di ritrasmissione del *keep-alive* (anche se questa non è una condizione molto probabile).

Come esempio dell'utilizzo di questa opzione introduciamo all'interno del nostro server per il servizio *echo* la nuova opzione `-k` che permette di attivare il *keep-alive* sui socket; tralasciando la parte relativa alla gestione di detta opzione (che si limita ad assegnare ad 1 la variabile `keepalive`) tutte le modifiche al server sono riportate in fig. 16.12. Al solito il codice completo è contenuto nel file `TCP_echo4_fourth.c` dei sorgenti allegati alla guida.

Come si può notare la variabile `keepalive` è preimpostata (8) ad un valore nullo; essa viene utilizzata sia come variabile logica per la condizione (14) che controlla l'attivazione del *keep-alive* che come valore dell'argomento `optval` della chiamata a `setsockopt` (16). A seconda del suo valore tutte le volte che un processo figlio viene eseguito in risposta ad una connessione verrà pertanto eseguita o meno la sezione (14-17) che esegue l'impostazione di `SO_KEEPALIVE` sul socket connesso, attivando il relativo comportamento.

Le opzioni `SO_REUSEADDR` e `SO_REUSEPORT`

La seconda opzione da approfondire è `SO_REUSEADDR`, che consente di eseguire `bind` su un socket anche quando la porta specificata è già in uso da parte di un altro socket. Si ricordi infatti che, come accennato in sez. 15.2.1, normalmente la funzione `bind` fallisce con un errore di `EADDRINUSE` se la porta scelta è già utilizzata da un altro socket, proprio per evitare che possano essere lanciati due server sullo stesso indirizzo e la stessa porta, che verrebbero a contendersi i pacchetti aventi quella destinazione.

Esistono però situazioni ed esigenze particolari in cui non si vuole che questo comportamento di salvaguardia accada, ed allora si può fare ricorso a questa opzione. La questione è comunque abbastanza complessa in quanto, come sottolinea Stevens in [?], si distinguono ben quattro casi diversi in cui è prevista la possibilità di un utilizzo di questa opzione, il che la rende una delle più difficili da capire.

Il primo caso in cui si fa ricorso a `SO_REUSEADDR`, che è anche il più comune, è quello in cui un server è terminato ma esistono ancora dei processi figli che mantengono attiva almeno una connessione remota che utilizza l'indirizzo locale, mantenendo occupata la porta. Quando si riesegue il server allora questo riceve un errore sulla chiamata a `bind` dato che la porta è ancora utilizzata in una connessione esistente.²⁵ Inoltre se si usa il protocollo TCP questo può avvenire anche dopo tutti i processi figli sono terminati, dato che una connessione può restare attiva anche dopo la chiusura del socket, mantenendosi nello stato `TIME_WAIT` (vedi sez. 15.1.5).

Usando `SO_REUSEADDR` fra la chiamata a `socket` e quella a `bind` si consente a quest'ultima di avere comunque successo anche se la connessione è attiva (o nello stato `TIME_WAIT`). È bene però ricordare (si riveda quanto detto in sez. 15.1.5) che la presenza dello stato `TIME_WAIT` ha una ragione, ed infatti se si usa questa opzione esiste sempre una probabilità, anche se estre-

²⁵questa è una delle domande più frequenti relative allo sviluppo, in quanto è piuttosto comune trovarsi in questa situazione quando si sta sviluppando un server che si ferma e si riavvia in continuazione dopo aver fatto modifiche.

```

1 int sockbindopt(char *host, char *serv, int prot, int type, int reuse)
2 {
3     struct addrinfo hint, *addr, *save;
4     int res;
5     int sock;
6     char buf[INET6_ADDRSTRLEN];
7     ...
8     while (addr != NULL) {                                /* loop on possible addresses */
9         /* get a socket */
10        sock = socket(addr->ai_family, addr->ai_socktype, addr->ai_protocol);
11        ...
12        /* connect the socket */
13        if (setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
14                    &reuse, sizeof(reuse))) {
15            printf("error_on_socket_options\n");
16            return -1;
17        }
18        ...
19
20    return sock;
21 }

```

Figura 16.13: Le sezioni della funzione `sockbindopt` modificate rispetto al codice della precedente `sockbind`.

mamente remota,²⁶ che eventuali pacchetti rimasti intrappolati in una precedente connessione possano finire fra quelli di una nuova.

Come esempio di uso di questa connessione abbiamo predisposto una nuova versione della funzione `sockbind` (vedi fig. 16.10) che consenta l'impostazione di questa opzione. La nuova funzione è `sockbindopt`, e le principali differenze rispetto alla precedente sono illustrate in fig. 16.13, dove si sono riportate le sezioni di codice modificate rispetto alla versione precedente. Il codice completo della funzione si trova, insieme alle altre funzioni di servizio dei socket, all'interno del file `SocketUtils.c` dei sorgenti allegati alla guida.

In realtà tutto quello che si è fatto è stato introdurre nella nuova funzione (1) un nuovo argomento intero, `reuse`, che conterrà il valore logico da usare nella successiva chiamata (14) a `setsockopt`. Si è poi aggiunta una sezione (13-17) che esegue l'impostazione dell'opzione fra la chiamata a `socket` e quella a `bind`.

A questo punto basterà modificare il server per utilizzare la nuova funzione; in fig. 16.14 abbiamo riportato le sezioni modificate rispetto alla precedente versione di fig. 16.11. Al solito il codice completo è coi sorgenti allegati alla guida, nel file `TCP_echod_fifth.c`.

Anche in questo caso si è introdotta (8) una nuova variabile `reuse` che consente di controllare l'uso dell'opzione e che poi sarà usata (14) come ultimo argomento di `setsockopt`. Il valore di default di questa variabile è nullo, ma usando l'opzione `-r` nell'invocazione del server (al solito la gestione delle opzioni non è riportata in fig. 16.14) se ne potrà impostare ad 1 il valore, per cui in tal caso la successiva chiamata (13-17) a `setsockopt` attiverà l'opzione `SO_REUSEADDR`.

Il secondo caso in cui viene usata `SO_REUSEADDR` è quando si ha una macchina cui sono assegnati diversi indirizzi IP (o come suol dirsi *multi-homed*) e si vuole porre in ascolto sulla stessa porta un programma diverso (o una istanza diversa dello stesso programma) per indirizzi IP diversi. Si ricordi infatti che è sempre possibile indicare a `bind` di collegarsi solo su di un

²⁶perché ciò avvenga infatti non solo devono coincidere gli indirizzi IP e le porte degli estremi della nuova connessione, ma anche i numeri di sequenza dei pacchetti, e questo è estremamente improbabile.

```

1 int main(int argc, char *argv[])
2 {
3     /*
4     * Variables definition
5     */
6     int list_fd, conn_fd;
7     int keepalive = 0;
8     int reuse = 0;
9     ...
10    /* create and bind socket */
11    if ( (list_fd = sockbindopt(argv[optind], "echo", 6,
12                               SOCK_STREAM, reuse)) < 0) {
13        return 1;
14    }
15    ...
16    /* normal exit, never reached */
17    exit(0);
18 }

```

Figura 16.14: Il nuovo codice per l'apertura passiva del server *echo* che usa la nuova funzione `sockbindopt`.

indirizzo specifico; in tal caso se un altro programma cerca di riutilizzare la stessa porta (anche specificando un indirizzo diverso) otterrà un errore, a meno di non aver preventivamente impostato `SO_REUSEADDR`.

Usando questa opzione diventa anche possibile eseguire `bind` sull'indirizzo generico, e questo permetterà il collegamento per tutti gli indirizzi (di quelli presenti) per i quali la porta non risulti occupata da una precedente chiamata più specifica. Si tenga presente infatti che con il protocollo TCP non è in genere possibile far partire più server che eseguano `bind` sullo stesso indirizzo e la stessa porta se su di esso c'è già un socket in ascolto, cioè ottenere quello che viene chiamato un *completely duplicate binding* (per questo è stata introdotta `SO_REUSEPORT`).

Il terzo impiego è simile al precedente e prevede l'uso di `bind` all'interno dello stesso programma per associare indirizzi locali diversi a socket diversi. In genere questo viene fatto per i socket UDP quando è necessario ottenere l'indirizzo a cui sono rivolte le richieste del client ed il sistema non supporta l'opzione `IP_RECVSTADDR`;²⁷ in tale modo si può sapere a quale socket corrisponde un certo indirizzo. Non ha senso fare questa operazione per un socket TCP dato che su di essi si può sempre invocare `getsockname` una volta che si è completata la connessione.

Infine il quarto caso è quello in cui si vuole effettivamente ottenere un *completely duplicate binding*, quando cioè si vuole eseguire `bind` su un indirizzo ed una porta che sono già "legati" ad un altro socket. Come accennato questo con TCP non è possibile, ed ha anche poco senso pensare di mettere in ascolto due server sulla stessa porta. Se però si prende in considerazione il traffico in *multicast*, diventa assolutamente normale che i pacchetti provenienti dal traffico in *multicast* possano essere ricevuti da più applicazioni o da diverse istanze della stessa applicazione sulla stessa porte di un indirizzo di *multicast*.

Un esempio classico è quello di uno streaming di dati (audio, video, ecc.) in cui l'uso del *multicast* consente di trasmettere un solo pacchetto da recapitare a tutti i possibili destinatari (invece di inviarne un duplicato a ciascuno); in questo caso è perfettamente logico aspettarsi

²⁷nel caso di Linux questa opzione è stata supportata per un certo periodo nello sviluppo del kernel 2.1.x, ma è in seguito stata soppiantata dall'uso di `IP_PKTINFO` (vedi sez. 16.2.4).

che sulla stessa macchina più utenti possano lanciare un programma che permetta loro di ricevere gli stessi dati, e quindi effettuare un *completely duplicate binding*.

In questo caso utilizzando `SO_REUSEADDR` si può consentire ad una applicazione eseguire `bind` sulla stessa porta ed indirizzo usata da un'altra, così che anche essa possa ricevere gli stessi pacchetti. Come detto la cosa non è possibile con i socket TCP (per i quali il *multicast* comunque non è applicabile), ma lo è per quelli UDP che è il protocollo normalmente in uso da parte di queste applicazioni. La regola è che quando si hanno più applicazioni che hanno eseguito `bind` sulla stessa porta, di tutti pacchetti destinati ad un indirizzo di *broadcast* o di *multicast* viene inviata una copia a ciascuna applicazione. Non è definito invece cosa accade qualora il pacchetto sia destinato ad un indirizzo normale (*unicast*).

Esistono però dei casi, in particolare per l'uso di programmi *multithreaded*, in cui può servire un *completely duplicate binding* anche per delle ordinarie connessioni TCP. Per supportare queste esigenze a partire dal kernel 3.9 è stata introdotta un'altra opzione, `SO_REUSEPORT` (già presente in altri sistemi come BSD), che consente di eseguire il *completely duplicate binding*, fintanto che essa venga specificata per tutti i socket interessati. Come per `SO_REUSEADDR` sarà possibile usare l'opzione su un socket legato allo stesso indirizzo e porta solo se il programma che ha eseguito il primo `bind` ha impostato questa opzione.

Nel caso di `SO_REUSEPORT` oltre al fatto che l'opzione deve essere attivata sul socket prima di chiamare `bind` ed attiva su tutti i socket con *completely duplicate binding*, è richiesto pure che tutti i processi che si mettono in ascolto sullo stesso indirizzo e porta abbiano lo stesso *UID* effettivo, per evitare che un altro utente possa ottenere il relativo traffico (eseguendo quello che viene definito *port hijacking* o *port stealing*). L'opzione utilizza per `optval` un intero usato come valore logico.

L'opzione si usa sia per socket TCP che UDP, nel primo caso consente un uso distribuito di `accept` in una applicazione *multithreaded* passando un diverso *listening socket* ad ogni thread, cosa che migliora le prestazioni rispetto all'approccio tradizionale di usare un thread per usare `accept` e distribuire le connessioni agli altri o avere più thread che competono per usare `accept` sul socket. Nel caso di UDP l'opzione consente di distribuire meglio i pacchetti su più processi o thread, rispetto all'approccio tradizionale di far competere gli stessi per l'accesso in ricezione al socket.

L'opzione `SO_LINGER`

La terza opzione da approfondire è `SO_LINGER`; essa, come il nome suggerisce, consente di “indugiare” nella chiusura di un socket. Il comportamento standard sia di `close` che `shutdown` è infatti quello di terminare immediatamente dopo la chiamata, mentre il procedimento di chiusura della connessione (o di un lato di essa) ed il rispettivo invio sulla rete di tutti i dati ancora presenti nei buffer, viene gestito in sottofondo dal kernel.

```
struct linger
{
    int l_onoff;           /* Nonzero to linger on close.  */
    int l_linger;         /* Time to linger (in seconds). */
}
```

Figura 16.15: La struttura `linger` richiesta come valore dell'argomento `optval` per l'impostazione dell'opzione dei socket `SO_LINGER`.

L'uso di `SO_LINGER` con `setsockopt` permette di modificare (ed eventualmente ripristinare) questo comportamento in base ai valori passati nei campi della struttura `linger`, illustrata in fig. 16.15. Fintanto che il valore del campo `l_onoff` di `linger` è nullo la modalità che viene impostata (qualunque sia il valore di `l_linger`) è quella standard appena illustrata; questa combinazione viene utilizzata per riportarsi al comportamento normale qualora esso sia stato cambiato da una precedente chiamata.

Se si utilizza un valore di `l_onoff` diverso da zero, il comportamento alla chiusura viene a dipendere dal valore specificato per il campo `l_linger`; se quest'ultimo è nullo l'uso delle funzioni `close` e `shutdown` provoca la terminazione immediata della connessione: nel caso di TCP cioè non viene eseguito il procedimento di chiusura illustrato in sez. 15.1.3, ma tutti i dati ancora presenti nel buffer vengono immediatamente scartati e sulla rete viene inviato un segmento di RST che termina immediatamente la connessione.

Un esempio di questo comportamento si può abilitare nel nostro client del servizio *echo* utilizzando l'opzione `-r`; riportiamo in fig. 16.16 la sezione di codice che permette di introdurre questa funzionalità; al solito il codice completo è disponibile nei sorgenti allegati.

```

1  ...
2  /* check if resetting on close is required */
3  if (reset) {
4      printf("Setting reset on close\n");
5      ling.l_onoff = 1;
6      ling.l_linger = 0;
7      if (setsockopt(sock, SOL_SOCKET, SO_LINGER, &ling, sizeof(ling))) {
8          perror("Cannot set linger");
9          exit(1);
10     }
11 }
12 ...

```

Figura 16.16: La sezione del codice del client *echo* che imposta la terminazione immediata della connessione in caso di chiusura.

La sezione indicata viene eseguita dopo aver effettuato la connessione e prima di chiamare la funzione di gestione, cioè fra le righe (12) e (13) del precedente esempio di fig. 16.9. Il codice si limita semplicemente a controllare (3) il valore della variabile `reset` che assegnata nella gestione delle opzioni in corrispondenza all'uso di `-r` nella chiamata del client. Nel caso questa sia diversa da zero vengono impostati (5-6) i valori della struttura `ling` che permettono una terminazione immediata della connessione. Questa viene poi usata nella successiva (7) chiamata a `setsockopt`. Al solito si controlla (7-10) il valore di ritorno e si termina il programma in caso di errore, stampandone il valore.

Infine l'ultima possibilità, quella in cui si utilizza effettivamente `SO_LINGER` per *indugiare* nella chiusura, è quella in cui sia `l_onoff` che `l_linger` hanno un valore diverso da zero. Se si esegue l'impostazione con questi valori sia `close` che `shutdown` si bloccano, nel frattempo viene eseguita la normale procedura di conclusione della connessione (quella di sez. 15.1.3) ma entrambe le funzioni non ritornano fintanto che non si sia concluso il procedimento di chiusura della connessione, o non sia passato un numero di secondi²⁸ pari al valore specificato in `l_linger`.

²⁸questa è l'unità di misura indicata da POSIX ed adottata da Linux, altri kernel possono usare unità di misura diverse, oppure usare il campo `l_linger` come valore logico (ignorandone il valore) per rendere (quando diverso da zero) `close` e `shutdown` bloccanti fino al completamento della trasmissione dei dati sul buffer.

16.2.4 Le opzioni per il protocollo IPv4

Il secondo insieme di opzioni dei socket che tratteremo è quello relativo ai socket che usano il protocollo IPv4; come per le precedenti opzioni generiche una descrizione di esse è disponibile nella settima sezione delle pagine di manuale, nel caso specifico la documentazione si può consultare con `man 7 ip`.

Opzione	get	set	flag	Tipo	Descrizione
IP_ADD_MEMBERSHIP		•		ip_mreqn	Si unisce a un gruppo di <i>multicast</i> .
IP_ADD_SOURCE_MEMBERSHIP		•		ip_mreq_source	Si unisce a un gruppo di <i>multicast</i> per una sorgente.
IP_BLOCK_SOURCE		•		ip_mreq_source	Smette di ricevere dati di <i>multicast</i> per una sorgente.
IP_DROP_MEMBERSHIP		•		ip_mreqn	Si sgancia da un gruppo di <i>multicast</i> .
IP_DROP_SOURCE_MEMBERSHIP		•		ip_mreq_source	Si sgancia da un gruppo di <i>multicast</i> per una sorgente.
IP_FREEBIND	•	•	•	int	Abilita il <i>binding</i> a un indirizzo IP non locale ancora non assegnato.
IP_HDRINCL	•	•	•	int	Passa l'intestazione di IP nei dati.
IP_MINTTL	•	•		int	Imposta il valore minimo del TTL per i pacchetti accettati.
IP_MSFILTER	•	•		ip_msfilter	Accesso completo all'interfaccia per il filtraggio delle sorgenti <i>multicast</i> .
IP_MTU	•			int	Legge il valore attuale della MTU.
IP_MTU_DISCOVER	•	•		int	Imposta il <i>Path MTU Discovery</i> .
IP_MULTICAST_ALL	•	•	•	int	Imposta l'interfaccia locale di un socket <i>multicast</i> .
IP_MULTICAST_IF		•		ip_mreqn	Imposta l'interfaccia locale di un socket <i>multicast</i> .
IP_MULTICAST_LOOP	•	•	•	int	Controlla il reinvio a se stessi dei dati di <i>multicast</i> .
IP_MULTICAST_TTL	•	•		int	Imposta il TTL per i pacchetti <i>multicast</i> .
IP_NODEFRAG	•	•	•	int	Disabilita il riassettaggio di pacchetti frammentati.
IP_OPTIONS	•	•		void *	Imposta o riceve le opzioni di IP.
IP_PKTINFO	•	•	•	int	Abilita i messaggi di informazione.
IP_RECVERR	•	•	•	int	Abilita i messaggi di errore affidabili.
IP_RECVOPTS	•	•	•	int	Passa un messaggio con le opzioni IP.
IP_RECVORIGSTADDR	•	•	•	int	Abilita i messaggi con l'indirizzo di destinazione originale.
IP_RECVTOS	•	•	•	int	Passa un messaggio col campo TOS.
IP_RECVTTL	•	•	•	int	Abilita i messaggi col campo TTL.
IP_RETOPTS	•	•	•	int	Abilita i messaggi con le opzioni IP non trattate.
IP_ROUTER_ALERT	•	•	•	int	Imposta l'opzione <i>IP router alert</i> sui pacchetti.
IP_TOS	•	•		int	Imposta il valore del campo TOS.
IP_TRANSPARENT	•	•	•	int	Abilita un <i>proxing</i> trasparente sul socket.
IP_TTL	•	•		int	Imposta il valore del campo TTL.
IP_UNBLOCK_SOURCE		•		ip_mreq_source	Ricomincia a ricevere dati di <i>multicast</i> per una sorgente.

Tabella 16.14: Le opzioni disponibili al livello IPPROTO_IP.

Se si vuole operare su queste opzioni generiche il livello da utilizzare è IPPROTO_IP (o l'equivalente SOL_IP); si è riportato un elenco di queste opzioni in tab. 16.14. Le costanti indicanti le opzioni, e tutte le altre costanti ad esse collegate, sono definite in `netinet/ip.h`,

ed accessibili includendo detto file.

Le descrizioni riportate in tab. 16.14 sono estremamente succinte, una maggiore quantità di dettagli sulle varie opzioni è fornita nel seguente elenco:

IP_ADD_MEMBERSHIP

L'opzione consente di unirsi ad gruppo di *multicast*, e può essere usata solo con `setsockopt`. L'argomento `optval` in questo caso deve essere una struttura di tipo `ip_mreqn`, illustrata in fig. 16.17, che permette di indicare, con il campo `imr_multiaddr` l'indirizzo del gruppo di *multicast* a cui ci si vuole unire, con il campo `imr_address` l'indirizzo dell'interfaccia locale con cui unirsi al gruppo di *multicast* e con `imr_ifindex` l'indice dell'interfaccia da utilizzare (un valore nullo indica una interfaccia qualunque).

```

struct ip_mreqn {
    struct in_addr imr_multiaddr; /* IP multicast group address */
    struct in_addr imr_address;   /* IP address of local interface */
    int           imr_ifindex;    /* interface index */
};

```

Figura 16.17: La struttura `ip_mreqn` utilizzata per unirsi a un gruppo di *multicast*.

Questa struttura è presente a partire dal kernel 2.2, per compatibilità è possibile utilizzare anche un argomento di tipo `ip_mreq`, presente fino dal kernel 1.2, che differisce da essa soltanto per l'assenza del campo `imr_ifindex`; il kernel riconosce il tipo di struttura in base alla differente dimensione passata in `optlen`.

IP_ADD_SOURCE_MEMBERSHIP

L'opzione consente di unirsi ad gruppo di *multicast*, ricevendo i dati solo da una sorgente specifica; come `IP_ADD_MEMBERSHIP` può essere usata solo con `setsockopt`.

```

struct ip_mreq_source {
    struct in_addr imr_multiaddr; /* IP multicast group address */
    struct in_addr imr_interface; /* IP address of local interface */
    struct in_addr imr_sourceaddr; /* IP address of multicast source */
};

```

Figura 16.18: La struttura `ip_mreqn` utilizzata per unirsi a un gruppo di *multicast* per una specifica sorgente.

L'argomento `optval` in questo caso è una struttura `ip_mreq_source` illustrata in fig. 16.18, dove il campo `imr_multiaddr` è l'indirizzo del gruppo di *multicast*, il campo `imr_interface` l'indirizzo dell'interfaccia locale che deve essere usata per aggiungersi al gruppo di *multicast* e il campo `imr_sourceaddr` l'indirizzo della sorgente da cui l'applicazione vuole ricevere i dati. L'opzione può essere ripetuta più volte per collegarsi a diverse sorgenti.

IP_BLOCK_SOURCE

Questa opzione, disponibile dal kernel 2.4.22, consente di smettere di ricevere dati di *multicast* dalla sorgente (e relativo gruppo) specificati dalla struttura

`ip_mreq_source` (vedi fig. 16.18) passata come argomento `optval`. L'opzione è utilizzabile solo se si è già registrati nel gruppo di *multicast* indicato con un precedente utilizzo di `IP_ADD_MEMBERSHIP` o `IP_ADD_SOURCE_MEMBERSHIP`.

IP_DROP_MEMBERSHIP

Lascia un gruppo di *multicast*, prende per `optval` la stessa struttura `ip_mreqn` (o `ip_mreq`) usata per `IP_ADD_MEMBERSHIP` (vedi fig. 16.17).

IP_DROP_SOURCE_MEMBERSHIP

Lascia un gruppo di *multicast* per una specifica sorgente, prende per `optval` la stessa struttura `ip_mreq_source` usata per `IP_ADD_SOURCE_MEMBERSHIP` (vedi fig. 16.18). Se ci si è registrati per più sorgenti nello stesso gruppo, si continuerà a ricevere dati sulle altre. Per smettere di ricevere dati da tutte le sorgenti occorre usare l'opzione `IP_LEAVE_GROUP`.

IP_FREEBIND Se abilitata questa opzione, disponibile dal kernel 2.4, consente di usare `bind` anche su un indirizzo IP non locale o che ancora non è stato assegnato. Questo permette ad una applicazione di mettersi in ascolto su un socket prima che l'interfaccia sottostante o l'indirizzo che questa deve usare sia stato configurato. È l'equivalente a livello di singolo socket dell'uso della *sysctl* `ip_nonlocal_bind` che vedremo in sez. 16.4.3. Prende per `optval` un intero usato come valore logico.

IP_HDRINCL Se viene abilitata questa opzione, presente dal kernel 2.0, l'utente deve fornire lui stesso l'intestazione del protocollo IP in testa ai propri dati. L'opzione è valida soltanto per socket di tipo `SOCK_RAW`, e quando utilizzata eventuali valori impostati con `IP_OPTIONS`, `IP_TOS` o `IP_TTL` sono ignorati. In ogni caso prima della spedizione alcuni campi dell'intestazione vengono comunque modificati dal kernel, torneremo sull'argomento in sez. 17.3.1. Prende per `optval` un intero usato come valore logico.

IP_MSFILTER L'opzione, introdotta con il kernel 2.4.22, fornisce accesso completo all'interfaccia per il filtraggio delle sorgenti di *multicast* (il cosiddetto *multicast source filtering*, definito dall'RFC 3376).

```

struct ip_msfilter {
    struct in_addr  imsf_multiaddr; /* IP multicast group address */
    struct in_addr  imsf_interface; /* IP address of local interface */
    uint32_t        imsf_fmode;     /* Filter-mode */
    uint32_t        imsf_numsrc;    /* Number of sources in the
                                     following array */
    struct in_addr  imsf_slist[1]; /* Array of source addresses */
};

```

Figura 16.19: La struttura `ip_msfilter` utilizzata per il *multicast source filtering*.

L'argomento `optval` deve essere una struttura di tipo `ip_msfilter` (illustrata in fig. 16.19); il campo `imsf_multiaddr` indica l'indirizzo del gruppo di *multicast*, il campo `imsf_interface` l'indirizzo dell'interfaccia locale, il campo `imsf_mode` indica la modalità di filtraggio e con i campi `imsf_numsrc` e `imsf_slist` rispettivamente la lunghezza della lista, e la lista stessa, degli indirizzi sorgente.

Come ausilio all'uso di questa opzione sono disponibili le macro `MCAST_INCLUDE` e `MCAST_EXCLUDE` che si possono usare per `imsf_mode`. Inoltre si può usare la macro `IP_MSFILTER_SIZE(n)` per determinare il valore di `optlen` con una struttura `ip_msfilter` contenente `n` sorgenti in `imsf_slist`.

IP_MINTTL L'opzione, introdotta con il kernel 2.6.34, imposta un valore minimo per il campo *Time to Live* dei pacchetti associati al socket su cui è attivata, che se non rispettato ne causa lo scarto automatico. L'opzione è nata per implementare l'RFC 5082 che la prevede come forma di protezione per i router che usano il protocollo BGP poiché questi, essendo in genere adiacenti, possono, impostando un valore di 255, scartare automaticamente tutti gli eventuali pacchetti falsi creati da un attacco a questo protocollo, senza doversi curare di verificarne la validità.²⁹

IP_MTU Permette di leggere il valore della *Path MTU* del socket. L'opzione richiede per `optval` un intero che conterrà il valore della *Path MTU* in byte. Questa è una opzione introdotta con i kernel della serie 2.2.x, ed è specifica di Linux.

È tramite questa opzione che un programma può leggere, quando si è avuto un errore di `EMSGSIZE`, il valore della MTU corrente del socket. Si tenga presente che per poter usare questa opzione, oltre ad avere abilitato la scoperta della *Path MTU*, occorre che il socket sia stato esplicitamente connesso con `connect`.

Ad esempio con i socket UDP si può ottenere una stima iniziale della *Path MTU* eseguendo prima una `connect` verso la destinazione, e poi usando `getsockopt` con questa opzione. Si può anche avviare esplicitamente il procedimento di scoperta inviando un pacchetto di grosse dimensioni (che verrà scartato) e ripetendo l'invio coi dati aggiornati. Si tenga infine conto che durante il procedimento i pacchetti iniziali possono essere perduti, ed è compito dell'applicazione gestirne una eventuale ritrasmissione.

IP_MTU_DISCOVER

Questa è una opzione introdotta con i kernel della serie 2.2.x, ed è specifica di Linux. L'opzione permette di scrivere o leggere le impostazioni della modalità usata per la determinazione della *Path MTU* (vedi sez. 13.3.5) del socket. L'opzione prende per `optval` un valore intero che indica la modalità usata, da specificare con una delle costanti riportate in tab. 16.15.

Valore	Significato
<code>IP_PMTUDISC_DONT</code>	0 Non effettua la ricerca dalla <i>Path MTU</i> .
<code>IP_PMTUDISC_WANT</code>	1 Utilizza il valore impostato per la rotta utilizzata dai pacchetti (dal comando <code>route</code>).
<code>IP_PMTUDISC_DO</code>	2 Esegue la procedura di determinazione della <i>Path MTU</i> come richiesto dall'RFC 1191.
<code>IP_PMTUDISC_PROBE</code>	? .

Tabella 16.15: Valori possibili per l'argomento `optval` di `IP_MTU_DISCOVER`.

²⁹l'attacco viene in genere portato per causare un *Denial of Service* aumentando il consumo di CPU del router nella verifica dell'autenticità di un gran numero di pacchetti falsi; questi, arrivando da sorgenti diverse da un router adiacente, non potrebbero più avere un TTL di 255 anche qualora questo fosse stato il valore di partenza, e l'impostazione dell'opzione consente di scartarli senza carico aggiuntivo sulla CPU (che altrimenti dovrebbe calcolare una checksum).

Il valore di default applicato ai socket di tipo `SOCK_STREAM` è determinato dal parametro `ip_no_pmtu_disc` (vedi sez. 16.4.1), mentre per tutti gli altri socket di default la ricerca è disabilitata ed è responsabilità del programma creare pacchetti di dimensioni appropriate e ritrasmettere eventuali pacchetti persi. Se l'opzione viene abilitata, il kernel si incaricherà di tenere traccia automaticamente della *Path MTU* verso ciascuna destinazione, e rifiuterà immediatamente la trasmissione di pacchetti di dimensioni maggiori della MTU con un errore di `EMSGSIZE`.³⁰

IP_MULTICAST_IF

Imposta l'interfaccia locale per l'utilizzo del *multicast*, ed utilizza come `optval` le stesse strutture `ip_mreqn` o `ip_mreq` delle due precedenti opzioni.

IP_MULTICAST_LOOP

L'opzione consente di decidere se i dati che si inviano su un socket usato con il *multicast* vengano ricevuti anche sulla stessa macchina da cui li si stanno inviando. Prende per `optval` un intero usato come valore logico.

In generale se si vuole che eventuali client possano ricevere i dati che si inviano occorre che questa funzionalità sia abilitata (come avviene di default). Qualora però non si voglia generare traffico per dati che già sono disponibili in locale l'uso di questa opzione permette di disabilitare questo tipo di traffico.

IP_MULTICAST_TTL

L'opzione permette di impostare o leggere il valore del campo TTL per i pacchetti *multicast* in uscita associati al socket. È importante che questo valore sia il più basso possibile, ed il default è 1, che significa che i pacchetti non potranno uscire dalla rete locale. Questa opzione consente ai programmi che lo richiedono di superare questo limite. L'opzione richiede per `optval` un intero che conterrà il valore del TTL.

IP_OPTIONS

L'opzione permette di impostare o leggere le opzioni del protocollo IP (si veda sez. B.1.3). L'opzione prende come valore dell'argomento `optval` un puntatore ad un buffer dove sono mantenute le opzioni, mentre `optlen` indica la dimensione di quest'ultimo. Quando la si usa con `getsockopt` vengono lette le opzioni IP utilizzate per la spedizione, quando la si usa con `setsockopt` vengono impostate le opzioni specificate. L'uso di questa opzione richiede una profonda conoscenza del funzionamento del protocollo, torneremo in parte sull'argomento in sez. 18.2.1.

IP_PKTINFO

Quando abilitata l'opzione permette di ricevere insieme ai pacchetti un messaggio ancillare (vedi sez. 18.1.3) di tipo `IP_PKTINFO` contenente una struttura `pktinfo` (vedi fig. 16.20) che mantiene una serie di informazioni riguardo i pacchetti in arrivo. In particolare è possibile conoscere l'interfaccia su cui è stato ricevuto un pacchetto (nel campo `ipi_ifindex`),³¹ l'indirizzo locale da esso utilizzato (nel campo `ipi_spec_dst`) e l'indirizzo remoto dello stesso (nel campo `ipi_addr`).

L'opzione è utilizzabile solo per socket di tipo `SOCK_DGRAM`. Questa è una opzione introdotta con i kernel della serie 2.2.x, ed è specifica di Linux;³² essa permette

³⁰in caso contrario la trasmissione del pacchetto sarebbe effettuata, ottenendo o un fallimento successivo della trasmissione, o la frammentazione dello stesso.

³¹in questo campo viene restituito il valore numerico dell'indice dell'interfaccia, sez. 16.3.2.

³²non dovrebbe pertanto essere utilizzata se si ha a cuore la portabilità.

```

struct in_pktinfo {
    unsigned int    ipi_ifindex;    /* Interface index */
    struct in_addr  ipi_spec_dst;    /* Local address */
    struct in_addr  ipi_addr;       /* Header Destination address */
};

```

Figura 16.20: La struttura `pktinfo` usata dall'opzione `IP_PKTINFO` per ricavare informazioni sui pacchetti di un socket di tipo `SOCK_DGRAM`.

di sostituire le opzioni `IP_RECVSTADDR` e `IP_RECVIF` presenti in altri Unix (la relativa informazione è quella ottenibile rispettivamente dai campi `ipi_addr` e `ipi_ifindex` di `pktinfo`).

L'opzione prende per `optval` un intero usato come valore logico, che specifica soltanto se insieme al pacchetto deve anche essere inviato o ricevuto il messaggio `IP_PKTINFO` (vedi sez. 18.1.3); il messaggio stesso dovrà poi essere letto o scritto direttamente con `recvmsg` e `sendmsg` (vedi sez. 18.1.2).

- IP_RECVERR** Questa è una opzione introdotta con i kernel della serie 2.2.x, ed è specifica di Linux. Essa permette di usufruire di un meccanismo affidabile per ottenere un maggior numero di informazioni in caso di errori. Se l'opzione è abilitata tutti gli errori generati su un socket vengono memorizzati su una coda, dalla quale poi possono essere letti con `recvmsg` (vedi sez. 18.1.2) come messaggi ancillari (torneremo su questo in sez. 18.1.3) di tipo `IP_RECVERR`. L'opzione richiede per `optval` un intero usato come valore logico e non è applicabile a socket di tipo `SOCK_STREAM`.
- IP_RECVOPTS** Quando abilitata l'opzione permette di ricevere insieme ai pacchetti un messaggio ancillare (vedi sez. 18.1.3) di tipo `IP_OPTIONS`, contenente le opzioni IP del protocollo (vedi sez. B.1.3). Le intestazioni di instradamento e le altre opzioni sono già riempite con i dati locali. L'opzione richiede per `optval` un intero usato come valore logico. L'opzione non è supportata per socket di tipo `SOCK_STREAM`.
- IP_RECVTOS** Quando abilitata l'opzione permette di ricevere insieme ai pacchetti un messaggio ancillare (vedi sez. 18.1.3) di tipo `IP_TOS`, che contiene un byte con il valore del campo *Type of Service* dell'intestazione IP del pacchetto stesso (vedi sez. B.1.2). Prende per `optval` un intero usato come valore logico.
- IP_RECVTTL** Quando abilitata l'opzione permette di ricevere insieme ai pacchetti un messaggio ancillare (vedi sez. 18.1.3) di tipo `IP_RECVTTL`, contenente un byte con il valore del campo *Time to Live* dell'intestazione IP (vedi sez. B.1.2). L'opzione richiede per `optval` un intero usato come valore logico. L'opzione non è supportata per socket di tipo `SOCK_STREAM`.
- IP_RETOPTS** Identica alla precedente `IP_RECVOPTS`, ma in questo caso restituisce i dati grezzi delle opzioni, senza che siano riempiti i capi di instradamento e le marche temporali. L'opzione richiede per `optval` un intero usato come valore logico. L'opzione non è supportata per socket di tipo `SOCK_STREAM`.
- IP_ROUTER_ALERT** Questa è una opzione introdotta con i kernel della serie 2.2.x, ed è specifica di Linux. Prende per `optval` un intero usato come valore logico. Se abilitata passa

tutti i pacchetti con l'opzione *IP Router Alert* (vedi sez. B.1.3) che devono essere inoltrati al socket corrente. Può essere usata soltanto per socket di tipo raw.

IP_TOS L'opzione consente di leggere o impostare il campo *Type of Service* dell'intestazione IP (per una trattazione più dettagliata, che riporta anche i valori possibili e le relative costanti di definizione si veda sez. B.1.2) che permette di indicare le priorità dei pacchetti. Se impostato il valore verrà mantenuto per tutti i pacchetti del socket; alcuni valori (quelli che aumentano la priorità) richiedono i privilegi di amministrazione con la *capability* CAP_NET_ADMIN.

Il campo TOS è di 8 bit e l'opzione richiede per *optval* un intero che ne contenga il valore. Sono definite anche alcune costanti che definiscono alcuni valori standardizzati per il *Type of Service*, riportate in tab. B.4, il valore di default usato da Linux è IPTOS_LOWDELAY, ma esso può essere modificato con le funzionalità del cosiddetto *Advanced Routing*. Si ricordi che la priorità dei pacchetti può essere impostata anche in maniera indipendente dal protocollo utilizzando l'opzione SO_PRIORITY illustrata in sez. 16.2.2.

IP_TTL L'opzione consente di leggere o impostare per tutti i pacchetti associati al socket il campo *Time to Live* dell'intestazione IP che indica il numero massimo di *hop* (passaggi da un router ad un altro) restanti al pacchetto (per una trattazione più estesa si veda sez. B.1.2). Il campo TTL è di 8 bit e l'opzione richiede che *optval* sia un intero, che ne conterrà il valore.

16.2.5 Le opzioni per i protocolli TCP e UDP

In questa sezione tratteremo le varie opzioni disponibili per i socket che usano i due principali protocolli di comunicazione del livello di trasporto; UDP e TCP.³³ Dato che questi due protocolli sono entrambi trasportati su IP,³⁴ oltre alle opzioni generiche di sez. 16.2.2 saranno comunque disponibili anche le precedenti opzioni di sez. 16.2.4.³⁵

Il protocollo che supporta il maggior numero di opzioni è TCP; per poterle utilizzare occorre specificare SOL_TCP (o l'equivalente IPPROTO_TCP) come valore per l'argomento *level*. Si sono riportate le varie opzioni disponibili in tab. 16.16, dove sono elencate le rispettive costanti da utilizzare come valore per l'argomento *optname*. Dette costanti e tutte le altre costanti e strutture collegate all'uso delle opzioni TCP sono definite in *netinet/tcp.h*, ed accessibili includendo detto file.³⁶

Le descrizioni delle varie opzioni riportate in tab. 16.16 sono estremamente sintetiche ed indicative, la spiegazione del funzionamento delle singole opzioni con una maggiore quantità di dettagli è fornita nel seguente elenco:

TCP_NODELAY

il protocollo TCP utilizza un meccanismo di bufferizzazione dei dati uscenti, per evitare la trasmissione di tanti piccoli segmenti con un utilizzo non ottimale della

³³come per le precedenti, una descrizione di queste opzioni è disponibile nella settima sezione delle pagine di manuale, che si può consultare rispettivamente con *man 7 tcp* e *man 7 udp*; le pagine di manuale però, alla stesura di questa sezione (Agosto 2006) sono alquanto incomplete.

³⁴qui si sottintende IPv4, ma le opzioni per TCP e UDP sono le stesse anche quando si usa IPv6.

³⁵in realtà in sez. 16.2.4 si sono riportate le opzioni per IPv4, al solito, qualora si stesse utilizzando IPv6, si potrebbero utilizzare le opzioni di quest'ultimo.

³⁶in realtà questo è il file usato dalle librerie; la definizione delle opzioni effettivamente supportate da Linux si trova nel file *include/linux/tcp.h* dei sorgenti del kernel, dal quale si sono estratte le costanti di tab. 16.16.

Opzione	get	set	flag	Tipo	Descrizione
TCP_NODELAY	●	●	●	int	Spedisce immediatamente i dati in segmenti singoli.
TCP_MAXSEG	●	●		int	Valore della MSS per i segmenti in uscita.
TCP_CORK	●	●	●	int	Accumula i dati in un unico segmento.
TCP_KEEPIIDLE	●	●		int	Tempo in secondi prima di inviare un <i>keepalive</i> .
TCP_KEEPINTVL	●	●		int	Tempo in secondi prima fra <i>keepalive</i> successivi.
TCP_KEEPCNT	●	●		int	Numero massimo di <i>keepalive</i> inviati.
TCP_SYNCNT	●	●		int	Numero massimo di ritrasmissioni di un SYN.
TCP_LINGER2	●	●		int	Tempo di vita in stato FIN_WAIT2.
TCP_DEFER_ACCEPT	●	●		int	Ritorna da accept solo in presenza di dati.
TCP_WINDOW_CLAMP	●	●		int	Valore della <i>advertised window</i> .
TCP_INFO	●			tcp_info	Restituisce informazioni sul socket.
TCP_QUICKACK	●	●	●	int	Abilita la modalità <i>quickack</i> .
TCP_CONGESTION	●	●		char *	Imposta l'algoritmo per il controllo della congestione.

Tabella 16.16: Le opzioni per i socket TCP disponibili al livello SOL_TCP.

banda disponibile.³⁷ Questo meccanismo è controllato da un apposito algoritmo (detto *algoritmo di Nagle*, vedi sez. ??). Il comportamento normale del protocollo prevede che i dati siano accumulati fintanto che non si raggiunge una quantità considerata adeguata per eseguire la trasmissione di un singolo segmento.

Ci sono però delle situazioni in cui questo comportamento può non essere desiderabile, ad esempio quando si sa in anticipo che l'applicazione invierà soltanto un piccolo quantitativo di dati;³⁸ in tal caso l'attesa introdotta dall'algoritmo di bufferizzazione non soltanto è inutile, ma peggiora le prestazioni introducendo un ritardo. Impostando questa opzione si disabilita l'uso dell'*algoritmo di Nagle* ed i dati vengono inviati immediatamente in singoli segmenti, qualunque sia la loro dimensione. Ovviamente l'uso di questa opzione è dedicato a chi ha esigenze particolari come quella illustrata, che possono essere stabilite solo per la singola applicazione.

Si tenga conto che questa opzione viene sovrascritta dall'eventuale impostazione dell'opzione TCP_CORK (il cui scopo è sostanzialmente l'opposto) che blocca l'invio immediato. Tuttavia quando la si abilita viene sempre forzato lo scaricamento della coda di invio (con conseguente trasmissione di tutti i dati pendenti), anche qualora si fosse già abilitata TCP_CORK.³⁹

TCP_MAXSEG

con questa opzione si legge o si imposta il valore della MSS (*Maximum Segment Size*, vedi sez. 13.3.5 e sez. ??) dei segmenti TCP uscenti. Se l'opzione è impostata prima di stabilire la connessione, si cambia anche il valore della MSS annunciata all'altro capo della connessione. Se si specificano valori maggiori della MTU questi verranno ignorati, inoltre TCP imporrà anche i suoi limiti massimo e minimo per questo valore.

TCP_CORK questa opzione è il complemento naturale di TCP_NODELAY e serve a gestire a livello applicativo la situazione opposta, cioè quella in cui si sa fin dal principio che si

³⁷il problema è chiamato anche *silly window syndrome*, per averne un'idea si pensi al risultato che si ottiene quando un programma di terminale invia un segmento TCP per ogni tasto premuto, 40 byte di intestazione di protocollo con 1 byte di dati trasmessi; per evitare situazioni del genere è stato introdotto l'*algoritmo di Nagle*.

³⁸è il caso classico di una richiesta HTTP.

³⁹si tenga presente però che TCP_CORK può essere specificata insieme a TCP_NODELAY soltanto a partire dal kernel 2.5.71.

dovranno inviare grosse quantità di dati. Anche in questo caso l'*algoritmo di Nagle* tenderà a suddividerli in dimensioni da lui ritenute opportune,⁴⁰ ma sapendo fin dall'inizio quale è la dimensione dei dati si potranno di nuovo ottenere delle migliori prestazioni disabilitandolo, e gestendo direttamente l'invio del nostro blocco di dati in soluzione unica.

Quando questa opzione viene abilitata non vengono inviati segmenti di dati fintanto che essa non venga disabilitata; a quel punto tutti i dati rimasti in coda saranno inviati in un solo segmento TCP. In sostanza con questa opzione si può controllare il flusso dei dati mettendo una sorta di “*tappo*” (da cui il nome in inglese) al flusso di uscita, in modo ottimizzare a mano l'uso della banda. Si tenga presente che per l'effettivo funzionamento ci si deve ricordare di disattivare l'opzione al termine dell'invio del blocco dei dati.

Si usa molto spesso TCP_CORK quando si effettua il trasferimento diretto di un blocco di dati da un file ad un socket con `sendfile` (vedi sez. 10.4.3), per inserire una intestazione prima della chiamata a questa funzione; senza di essa l'intestazione potrebbe venire spedita in un segmento a parte, che a seconda delle condizioni potrebbe richiedere anche una risposta di ACK, portando ad una notevole penalizzazione delle prestazioni.

Si tenga presente che l'implementazione corrente di TCP_CORK non consente di bloccare l'invio dei dati per più di 200 millisecondi, passati i quali i dati accumulati in coda fanno comunque. Questa opzione è tipica di Linux⁴¹ e non è disponibile su tutti i kernel unix-like, pertanto deve essere evitata se si vuole scrivere codice portabile.

TCP_KEEPIDLE

con questa opzione si legge o si imposta l'intervallo di tempo, in secondi, che deve trascorrere senza traffico sul socket prima che vengano inviati, qualora si sia attivata su di esso l'opzione SO_KEEPALIVE, i messaggi di *keep-alive* (si veda la trattazione relativa al *keep-alive* in sez. 16.2.3). Anche questa opzione non è disponibile su tutti i kernel unix-like e deve essere evitata se si vuole scrivere codice portabile.

TCP_KEEPINTVL

con questa opzione si legge o si imposta l'intervallo di tempo, in secondi, fra due messaggi di *keep-alive* successivi (si veda sempre quanto illustrato in sez. 16.2.3). Come la precedente non è disponibile su tutti i kernel unix-like e deve essere evitata se si vuole scrivere codice portabile.

TCP_KEEPCNT

con questa opzione si legge o si imposta il numero totale di messaggi di *keep-alive* da inviare prima di concludere che la connessione è caduta per assenza di risposte ad un messaggio di *keep-alive* (di nuovo vedi sez. 16.2.3). Come la precedente non è disponibile su tutti i kernel unix-like e deve essere evitata se si vuole scrivere codice portabile.

⁴⁰l'algoritmo cerca di tenere conto di queste situazioni, ma essendo un algoritmo generico tenderà comunque ad introdurre delle suddivisioni in segmenti diversi, anche quando potrebbero non essere necessarie, con conseguente spreco di banda.

⁴¹l'opzione è stata introdotta con i kernel della serie 2.4.x.

TCP_SYNCNT

con questa opzione si legge o si imposta il numero di tentativi di ritrasmissione dei segmenti SYN usati nel *three way handshake* prima che il tentativo di connessione venga abortito (si ricordi quanto accennato in sez. 15.2.2). Sovrascrive per il singolo socket il valore globale impostato con la *sysctl* `tcp_syn_retries` (vedi sez. 16.4.3). Non vengono accettati valori maggiori di 255; anche questa opzione non è standard e deve essere evitata se si vuole scrivere codice portabile.

TCP_LINGER2

con questa opzione si legge o si imposta, in numero di secondi, il tempo di sussistenza dei socket terminati nello stato `FIN_WAIT2` (si ricordi quanto visto in sez. 15.1.3).⁴² Questa opzione consente di sovrascrivere per il singolo socket il valore globale impostato con la *sysctl* `tcp_fin_timeout` (vedi sez. 16.4.3). Anche questa opzione è da evitare se si ha a cuore la portabilità del codice.

TCP_DEFER_ACCEPT

questa opzione consente di modificare il comportamento standard del protocollo TCP nello stabilirsi di una connessione; se ricordiamo il meccanismo del *three way handshake* illustrato in fig. 15.1 possiamo vedere che in genere un client inizierà ad inviare i dati ad un server solo dopo l'emissione dell'ultimo segmento di ACK.

Di nuovo esistono situazioni (e la più tipica è quella di una richiesta HTTP) in cui sarebbe utile inviare immediatamente la richiesta all'interno del segmento con l'ultimo ACK del *three way handshake*; si potrebbe così risparmiare l'invio di un segmento successivo per la richiesta e il ritardo sul server fra la ricezione dell'ACK e quello della richiesta.

Se si invoca `TCP_DEFER_ACCEPT` su un socket dal lato client (cioè dal lato da cui si invoca `connect`) si istruisce il kernel a non inviare immediatamente l'ACK finale del *three way handshake*, attendendo per un po' di tempo la prima scrittura, in modo da inviare i dati di questa insieme col segmento ACK. Chiaramente la correttezza di questo comportamento dipende in maniera diretta dal tipo di applicazione che usa il socket; con HTTP, che invia una breve richiesta, permette di risparmiare un segmento, con FTP, in cui invece si attende la ricezione del prompt del server, introduce un inutile ritardo.

Allo stesso tempo il protocollo TCP prevede che sul lato del server la funzione `accept` ritorni dopo la ricezione dell'ACK finale, in tal caso quello che si fa usualmente è lanciare un nuovo processo per leggere i successivi dati, che si bloccherà su una `read` se questi non sono disponibili; in questo modo si saranno impiegate delle risorse (per la creazione del nuovo processo) che non vengono usate immediatamente. L'uso di `TCP_DEFER_ACCEPT` consente di intervenire anche in questa situazione; quando la si invoca sul lato server (vale a dire su un socket in ascolto) l'opzione fa sì che `accept` ritorni soltanto quando sono presenti dei dati sul socket, e non alla ricezione dell'ACK conclusivo del *three way handshake*.

L'opzione prende un valore intero che indica il numero massimo di secondi per cui mantenere il ritardo, sia per quanto riguarda il ritorno di `accept` su un server, che per l'invio dell'ACK finale insieme ai dati su un client. L'opzione è specifica di Linux non deve essere utilizzata in codice che vuole essere portabile.⁴³

⁴²si tenga ben presente che questa opzione non ha nulla a che fare con l'opzione `SO_LINGER` che abbiamo visto in sez. 16.2.3.

⁴³su FreeBSD è presente una opzione `SO_ACCEPTFILTER` che consente di ottenere lo stesso comportamento di `TCP_DEFER_ACCEPT` per quanto riguarda il lato server.

TCP_WINDOW_CLAMP

con questa opzione si legge o si imposta alla dimensione specificata, in byte, il valore dichiarato della *advertised window* (vedi sez. ??). Il kernel impone comunque una dimensione minima pari a `SOCK_MIN_RCVBUF/2`. Questa opzione non deve essere utilizzata in codice che vuole essere portabile.

```

struct tcp_info
{
    u_int8_t      tcpi_state;
    u_int8_t      tcpi_ca_state;
    u_int8_t      tcpi_retransmits;
    u_int8_t      tcpi_probes;
    u_int8_t      tcpi_backoff;
    u_int8_t      tcpi_options;
    u_int8_t      tcpi_snd_wscale : 4, tcpi_rcv_wscale : 4;
    u_int32_t     tcpi_rto;
    u_int32_t     tcpi_ato;
    u_int32_t     tcpi_snd_mss;
    u_int32_t     tcpi_rcv_mss;
    u_int32_t     tcpi_unacked;
    u_int32_t     tcpi_sacked;
    u_int32_t     tcpi_lost;
    u_int32_t     tcpi_retrans;
    u_int32_t     tcpi_fackets;
    /* Times. */
    u_int32_t     tcpi_last_data_sent;
    u_int32_t     tcpi_last_ack_sent;      /* Not remembered, sorry.
*/
    u_int32_t     tcpi_last_data_recv;
    u_int32_t     tcpi_last_ack_recv;
    /* Metrics. */
    u_int32_t     tcpi_pmtu;
    u_int32_t     tcpi_rcv_ssthresh;
    u_int32_t     tcpi_rtt;
    u_int32_t     tcpi_rttvar;
    u_int32_t     tcpi_snd_ssthresh;
    u_int32_t     tcpi_snd_cwnd;
    u_int32_t     tcpi_advmss;
    u_int32_t     tcpi_reordering;
};

```

Figura 16.21: La struttura `tcp_info` contenente le informazioni sul socket restituita dall'opzione `TCP_INFO`.

TCP_INFO questa opzione, specifica di Linux, ma introdotta anche in altri kernel (ad esempio FreeBSD) permette di controllare lo stato interno di un socket TCP direttamente da un programma in *user space*. L'opzione restituisce in una speciale struttura `tcp_info`, la cui definizione è riportata in fig. 16.21, tutta una serie di dati che il kernel mantiene, relativi al socket. Anche questa opzione deve essere evitata se si vuole scrivere codice portabile.

Con questa opzione diventa possibile ricevere una serie di informazioni relative ad un socket TCP così da poter effettuare dei controlli senza dover passare attraverso delle operazioni di lettura. Ad esempio si può verificare se un socket è stato chiuso usando una funzione analoga a quella illustrata in fig. 16.22, in cui si utilizza il valore del campo `tcpi_state` di `tcp_info` per controllare lo stato del socket.

```

1 int is_closing(int sock)
2 {
3     struct tcp_info info;
4     socklen_t len = sizeof(info);
5     if (getsockopt(sock, SOL_TCP, TCP_INFO, &info, &len) != -1) {
6         if (info.tcpi_state == TCP_CLOSE ||
7             info.tcpi_state == TCP_CLOSE_WAIT ||
8             info.tcpi_state == TCP_CLOSING) {
9             return 1;
10        } else {
11            return 0;
12        }
13    } else {
14        return errno;
15    }
16 }

```

Figura 16.22: Codice della funzione `is_closing.c`, che controlla lo stato di un socket TCP per verificare se si sta chiudendo.

TCP_QUICKACK

con questa opzione è possibile eseguire una forma di controllo sull'invio dei segmenti ACK all'interno di in flusso di dati su TCP. In genere questo invio viene gestito direttamente dal kernel, il comportamento standard, corrispondente la valore logico di vero (in genere 1) per questa opzione, è quello di inviare immediatamente i segmenti ACK, in quanto normalmente questo significa che si è ricevuto un blocco di dati e si può passare all'elaborazione del blocco successivo.

Qualora però la nostra applicazione sappia in anticipo che alla ricezione di un blocco di dati seguirà immediatamente l'invio di un altro blocco,⁴⁴ poter accorpare quest'ultimo al segmento ACK permette di risparmiare sia in termini di dati inviati che di velocità di risposta. Per far questo si può utilizzare TCP_QUICKACK impostando un valore logico falso (cioè 0), in questo modo il kernel attenderà così da inviare il prossimo segmento di ACK insieme ai primi dati disponibili.

Si tenga presente che l'opzione non è permanente, vale a dire che una volta che la si sia impostata a 0 il kernel la riporterà al valore di default dopo il suo primo utilizzo. Sul lato server la si può impostare anche una volta sola su un socket in ascolto, ed essa verrà ereditata da tutti i socket che si otterranno da esso al ritorno di `accept`.

TCP_CONGESTION

questa opzione permette di impostare quale algoritmo per il controllo della congestione⁴⁵ utilizzare per il singolo socket. L'opzione è stata introdotta con il kernel 2.6.13,⁴⁶ e prende come per `optval` il puntatore ad un buffer contenente il nome dell'algoritmo di controllo che si vuole usare.

⁴⁴caso tipico ad esempio delle risposte alle richieste HTTP.

⁴⁵il controllo della congestione è un meccanismo previsto dal protocollo TCP (vedi sez. ??) per evitare di trasmettere inutilmente dati quando una connessione è congestionata; un buon algoritmo è fondamentale per il funzionamento del protocollo, dato che i pacchetti persi andrebbero ritrasmessi, per cui inviare un pacchetto su una linea congestionata potrebbe causare facilmente un peggioramento della situazione.

⁴⁶alla data di stesura di queste note (Set. 2006) è pure scarsamente documentata, tanto che non è neanche definita nelle intestazioni della *glibc* per cui occorre definirla a mano al suo valore che è 13.

L'uso di un nome anziché di un valore numerico è dovuto al fatto che gli algoritmi di controllo della congestione sono realizzati attraverso altrettanti moduli del kernel, e possono pertanto essere attivati a richiesta; il nome consente di caricare il rispettivo modulo e di introdurre moduli aggiuntivi che implementino altri meccanismi.

Per poter disporre di questa funzionalità occorre aver compilato il kernel attivando l'opzione di configurazione generale `TCP_CONG_ADVANCED`,⁴⁷ e poi abilitare i singoli moduli voluti con le varie `TCP_CONG_*` presenti per i vari algoritmi disponibili; un elenco di quelli attualmente supportati nella versione ufficiale del kernel è riportato in tab. 16.17.⁴⁸

Si tenga presente che prima della implementazione modulare alcuni di questi algoritmi erano disponibili soltanto come caratteristiche generali del sistema, attivabili per tutti i socket, questo è ancora possibile con la `sysctl tcp_congestion_control` (vedi sez. 16.4.3) che ha sostituito le precedenti `sysctl`.⁴⁹

Nome	Configurazione	Riferimento
reno	–	Algoritmo tradizionale, usato in caso di assenza degli altri.
bic	TCP_CONG_BIC	http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/index.htm .
cubic	TCP_CONG_CUBIC	http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/index.htm .
highspeed	TCP_CONG_HSTCP	http://www.icir.org/floyd/hstcp.html .
htcp	TCP_CONG_HTCP	http://www.hamilton.ie/net/htcp/ .
hybla	TCP_CONG_HYBLA	http://www.danielinux.net/projects.html .
scalable	TCP_CONG_SCALABLE	http://www.deneholme.net/tom/scalable/ .
vegas	TCP_CONG_VEGAS	http://www.cs.arizona.edu/protocols/ .
westwood	TCP_CONG_WESTWOOD	http://www.cs.ucla.edu/NRL/hpi/tcpw/ .

Tabella 16.17: Gli algoritmi per il controllo della congestione disponibili con Linux con le relative opzioni di configurazione da attivare.

Il protocollo UDP, anche per la sua maggiore semplicità, supporta un numero ridotto di opzioni, riportate in tab. 16.18; anche in questo caso per poterle utilizzare occorrerà impostare l'opportuno valore per l'argomento `level`, che è `SOL_UDP` (o l'equivalente `IPPROTO_UDP`). Le costanti che identificano dette opzioni sono definite in `netinet/udp.h`, ed accessibili includendo detto file.⁵⁰

Opzione	get	set	flag	Tipo	Descrizione
UDP_CORK	•	•	•	int	Accumula tutti i dati su un unico pacchetto.
UDP_ENCAP	•	•	•	int	Non documentata.

Tabella 16.18: Le opzioni per i socket UDP disponibili al livello `SOL_UDP`.

Ancora una volta le descrizioni contenute tab. 16.18 sono un semplice riferimento, una maggiore quantità di dettagli sulle caratteristiche delle opzioni citate è quello dell'elenco seguente:

UDP_CORK questa opzione ha l'identico effetto dell'analogo `TCP_CORK` vista in precedenza per il protocollo TCP, e quando abilitata consente di accumulare i dati in uscita su

⁴⁷disponibile come *TCP: advanced congestion control* nel menù *Network->Networking options*, che a sua volta renderà disponibile un ulteriore menù con gli algoritmi presenti.

⁴⁸la lista è presa dalla versione 2.6.17.

⁴⁹riportate anche, alla data di stesura di queste pagine (Set. 2006) nelle pagine di manuale, ma non più presenti.

⁵⁰come per TCP, la definizione delle opzioni effettivamente supportate dal kernel si trova in realtà nel file `include/linux/udp.h`, dal quale si sono estratte le costanti di tab. 16.18.

un solo pacchetto che verrà inviato una volta che la si disabiliti. L'opzione è stata introdotta con il kernel 2.5.44, e non deve essere utilizzata in codice che vuole essere portabile.

UDP_ENCAP

Questa opzione permette di gestire l'incapsulazione dei dati nel protocollo UDP. L'opzione è stata introdotta con il kernel 2.5.67, e non è documentata. Come la precedente è specifica di Linux e non deve essere utilizzata in codice portabile.

16.3 La gestione attraverso le funzioni di controllo

Benché la maggior parte delle caratteristiche dei socket sia gestibile con le funzioni `setsockopt` e `getsockopt`, alcune proprietà possono essere impostate attraverso le funzioni `fcntl` e `ioctl` già trattate in sez. 5.2.5; in quell'occasione abbiamo parlato di queste funzioni esclusivamente nell'ambito della loro applicazione a file descriptor associati a dei file normali; qui tratteremo invece i dettagli del loro utilizzo con file descriptor associati a dei socket.

16.3.1 L'uso di `ioctl` e `fcntl` per i socket generici

Tratteremo in questa sezione le caratteristiche specifiche delle funzioni `ioctl` e `fcntl` quando esse vengono utilizzate con dei socket generici. Quanto già detto in precedenza sez. 5.2.5 continua a valere; quello che tratteremo qui sono le operazioni ed i comandi che sono validi, o che hanno significati peculiari, quando queste funzioni vengono applicate a dei socket generici.

Nell'elenco seguente si riportano i valori specifici che può assumere il secondo argomento della funzione `ioctl` (`request`, che indica il tipo di operazione da effettuare) quando essa viene applicata ad un socket generico. Nell'elenco si illustrerà anche, per ciascuna operazione, il tipo di dato usato come terzo argomento della funzione ed il significato che esso viene ad assumere. Dato che in caso di lettura questi dati vengono restituiti come *value result argument*, con queste operazioni il terzo argomento deve sempre essere passato come puntatore ad una variabile (o struttura) precedentemente allocata. Le costanti che identificano le operazioni sono le seguenti:

SIOCGSTAMP

restituisce il contenuto di una struttura `timeval` con la marca temporale dell'ultimo pacchetto ricevuto sul socket, questa operazione può essere utilizzata per effettuare delle misurazioni precise del tempo di andata e ritorno⁵¹ dei pacchetti sulla rete.

SIOCSPGRP

imposta il processo o il *process group* a cui inviare i segnali `SIGIO` e `SIGURG` quando viene completata una operazione di I/O asincrono o arrivano dei dati urgenti (*out-of-band*). Il terzo argomento deve essere un puntatore ad una variabile di tipo `pid_t`; un valore positivo indica direttamente il *PID* del processo, mentre un valore negativo indica (col valore assoluto) il *process group*. Senza privilegi di amministratore o la *capability* `CAP_KILL` si può impostare solo se stessi o il proprio *process group*.

SIOCGPGRP

legge le impostazioni presenti sul socket relativamente all'eventuale processo o *process group* cui devono essere inviati i segnali `SIGIO` e `SIGURG`. Come per `SIOCSPGRP`

⁵¹il *Round Trip Time* cui abbiamo già accennato in sez. 13.3.4.

l'argomento passato deve un puntatore ad una variabile di tipo `pid_t`, con lo stesso significato. Qualora non sia presente nessuna impostazione verrà restituito un valore nullo.

FIOASYNC Abilita o disabilita la modalità di I/O asincrono sul socket. Questo significa (vedi sez. 10.3.1) che verrà inviato il segnale di `SIGIO` (o quanto impostato con `F_SETSIG`, vedi sez. 5.2.5) in caso di eventi di I/O sul socket.

Nel caso dei socket generici anche `fcntl` prevede un paio di comandi specifici; in questo caso il secondo argomento (`cmd`, che indica il comando) può assumere i due valori `FIOGETOWN` e `FIOSETOWN`, mentre il terzo argomento dovrà essere un puntatore ad una variabile di tipo `pid_t`. Questi due comandi sono una modalità alternativa di eseguire le stesse operazioni (lettura o impostazione del processo o del gruppo di processo che riceve i segnali) che si effettuano chiamando `ioctl` con `SIOCGGRP` e `SIOCSPGRP`.

16.3.2 L'uso di `ioctl` per l'accesso ai dispositivi di rete

Benché non strettamente attinenti alla gestione dei socket, vale la pena di trattare qui l'interfaccia di accesso a basso livello ai dispositivi di rete che viene appunto fornita attraverso la funzione `ioctl`. Questa non è attinente a caratteristiche specifiche di un qualche protocollo, ma si applica a tutti i socket, indipendentemente da tipo e famiglia degli stessi, e permette di impostare e rilevare le funzionalità delle interfacce di rete.

```

struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short ifr_flags;
        int ifr_ifindex;
        int ifr_metric;
        int ifr_mtu;
        struct ifmap ifr_map;
        char ifr_slave[IFNAMSIZ];
        char ifr_newname[IFNAMSIZ];
        char * ifr_data;
    };
};

```

Figura 16.23: La struttura `ifreq` utilizzata dalle `ioctl` per le operazioni di controllo sui dispositivi di rete.

Tutte le operazioni di questo tipo utilizzano come terzo argomento di `ioctl` il puntatore ad una struttura `ifreq`, la cui definizione è illustrata in fig. 16.23. Normalmente si utilizza il primo campo della struttura, `ifr_name` per specificare il nome dell'interfaccia su cui si vuole operare (ad esempio `eth0`, `ppp0`, ecc.), e si inseriscono (o ricevono) i valori relativi alle diverse caratteristiche e funzionalità nel secondo campo, che come si può notare è definito come una `union` proprio in quanto il suo significato varia a secondo dell'operazione scelta.

Si tenga inoltre presente che alcune di queste operazioni (in particolare quelle che modificano le caratteristiche dell'interfaccia) sono privilegiate e richiedono i privilegi di amministra-

tore o la *capability* CAP_NET_ADMIN, altrimenti si otterrà un errore di EPERM. Le costanti che identificano le operazioni disponibili sono le seguenti:

SIOCGIFNAME

questa è l'unica operazione che usa il campo `ifr_name` per restituire un risultato, tutte le altre lo utilizzano per indicare l'interfaccia sulla quale operare. L'operazione richiede che si indichi nel campo `ifr_ifindex` il valore numerico dell'*indice* dell'interfaccia, e restituisce il relativo nome in `ifr_name`.

Il kernel infatti assegna ad ogni interfaccia un numero progressivo, detto appunto *interface index*, che è quello che effettivamente la identifica nelle operazioni a basso livello, il nome dell'interfaccia è soltanto una etichetta associata a detto *indice*, che permette di rendere più comprensibile l'indicazione dell'interfaccia all'interno dei comandi. Una modalità per ottenere questo valore è usare il comando `ip link`, che fornisce un elenco delle interfacce presenti ordinato in base a tale valore (riportato come primo campo).

SIOCGIFINDEX

restituisce nel campo `ifr_ifindex` il valore numerico dell'indice dell'interfaccia specificata con `ifr_name`, è in sostanza l'operazione inversa di SIOCGIFNAME.

SIOCGIFFLAGS

permette di ottenere nel campo `ifr_flags` il valore corrente dei flag dell'interfaccia specificata (con `ifr_name`). Il valore restituito è una maschera binaria i cui bit sono identificabili attraverso le varie costanti di tab. 16.19.

Flag	Significato
IFF_UP	L'interfaccia è attiva.
IFF_BROADCAST	L'interfaccia ha impostato un indirizzo di <i>broadcast</i> valido.
IFF_DEBUG	È attivo il flag interno di debug.
IFF_LOOPBACK	L'interfaccia è una interfaccia di <i>loopback</i> .
IFF_POINTOPOINT	L'interfaccia è associata ad un collegamento <i>punto-punto</i> .
IFF_RUNNING	L'interfaccia ha delle risorse allocate (non può quindi essere disattivata).
IFF_NOARP	L'interfaccia ha il protocollo ARP disabilitato o l'indirizzo del livello di rete non è impostato.
IFF_PROMISC	L'interfaccia è nel cosiddetto <i>modo promiscuo</i> , riceve cioè tutti i pacchetti che vede passare, compresi quelli non direttamente indirizzati a lei.
IFF_NOTRAILERS	Evita l'uso di <i>trailer</i> nei pacchetti.
IFF_ALLMULTI	Riceve tutti i pacchetti di <i>multicast</i> .
IFF_MASTER	L'interfaccia è il master di un bundle per il bilanciamento di carico.
IFF_SLAVE	L'interfaccia è uno slave di un bundle per il bilanciamento di carico.
IFF_MULTICAST	L'interfaccia ha il supporto per il <i>multicast</i> attivo.
IFF_PORTSEL	L'interfaccia può impostare i suoi parametri hardware (con l'uso di <code>ifmap</code>).
IFF_AUTOMEDIA	L'interfaccia è in grado di selezionare automaticamente il tipo di collegamento.
IFF_DYNAMIC	Gli indirizzi assegnati all'interfaccia vengono persi quando questa viene disattivata.

Tabella 16.19: Le costanti che identificano i vari bit della maschera binaria `ifr_flags` che esprime i flag di una interfaccia di rete.

SIOCSIFFLAGS

permette di impostare il valore dei flag dell'interfaccia specificata (sempre con `ifr_name`, non staremo a ripeterlo oltre) attraverso il valore della maschera binaria da passare nel campo `ifr_flags`, che può essere ottenuta con l'OR aritmetico delle costanti di tab. 16.19; questa operazione è privilegiata.

SIOCGIFMETRIC

permette di leggere il valore della metrica del dispositivo associato all'interfaccia specificata nel campo `ifr_metric`. Attualmente non è implementato, e l'operazione restituisce sempre un valore nullo.

SIOCSIFMETRIC

permette di impostare il valore della metrica del dispositivo al valore specificato nel campo `ifr_metric`, attualmente non ancora implementato, restituisce un errore di `EOPNOTSUPP`.

SIOCGIFMTU

permette di leggere il valore della *Maximum Transfer Unit* del dispositivo nel campo `ifr_mtu`.

SIOCSIFMTU

permette di impostare il valore della *Maximum Transfer Unit* del dispositivo al valore specificato campo `ifr_mtu`. L'operazione è privilegiata, e si tenga presente che impostare un valore troppo basso può causare un blocco del kernel.

SIOCGIFHWADDR

permette di leggere il valore dell'indirizzo hardware del dispositivo associato all'interfaccia nel campo `ifr_hwaddr`; questo viene restituito come struttura `sockaddr` in cui il campo `sa_family` contiene un valore `ARPHRD_*` indicante il tipo di indirizzo ed il campo `sa_data` il valore binario dell'indirizzo hardware a partire dal byte 0.

SIOCSIFHWADDR

permette di impostare il valore dell'indirizzo hardware del dispositivo associato all'interfaccia attraverso il valore della struttura `sockaddr` (con lo stesso formato illustrato per `SIOCGIFHWADDR`) passata nel campo `ifr_hwaddr`. L'operazione è privilegiata.

SIOCSIFHWBROADCAST

imposta l'indirizzo *broadcast* hardware dell'interfaccia al valore specificato dal campo `ifr_hwaddr`. L'operazione è privilegiata.

SIOCGIFMAP

legge alcuni parametri hardware (memoria, interrupt, canali di DMA) del driver dell'interfaccia specificata, restituendo i relativi valori nel campo `ifr_map`; quest'ultimo contiene una struttura di tipo `ifmap`, la cui definizione è illustrata in fig. 16.24.

SIOCSIFMAP

imposta i parametri hardware del driver dell'interfaccia specificata, restituendo i relativi valori nel campo `ifr_map`. Come per `SIOCGIFMAP` questo deve essere passato come struttura `ifmap`, secondo la definizione di fig. 16.24.

SIOCADDMULTI

aggiunge un indirizzo di *multicast* ai filtri del livello di collegamento associati dell'interfaccia. Si deve usare un indirizzo hardware da specificare attraverso il campo

```

struct ifmap
{
    unsigned long    mem_start;
    unsigned long    mem_end;
    unsigned short   base_addr;
    unsigned char     irq;
    unsigned char     dma;
    unsigned char     port;
};

```

Figura 16.24: La struttura `ifmap` utilizzata per leggere ed impostare i valori dei parametri hardware di un driver di una interfaccia.

`ifr_hwaddr`, che conterrà l'opportuna struttura `sockaddr`; l'operazione è privilegiata. Per una modalità alternativa per eseguire la stessa operazione si possono usare i *packet socket*, vedi sez. 17.3.3.

SIOCDELMULTI

rimuove un indirizzo di *multicast* ai filtri del livello di collegamento dell'interfaccia, vuole un indirizzo hardware specificato come per `SIOCADDMULTI`. Anche questa operazione è privilegiata e può essere eseguita in forma alternativa con i *packet socket*.

SIOCGIFTXQLEN

permette di leggere la lunghezza della coda di trasmissione del dispositivo associato all'interfaccia specificata nel campo `ifr_qlen`.

SIOCSIFTXQLEN

permette di impostare il valore della lunghezza della coda di trasmissione del dispositivo associato all'interfaccia, questo deve essere specificato nel campo `ifr_qlen`. L'operazione è privilegiata.

SIOCSIFNAME

consente di cambiare il nome dell'interfaccia indicata da `ifr_name` utilizzando il nuovo nome specificato nel campo `ifr_rename`.

Una ulteriore operazione, che consente di ricavare le caratteristiche delle interfacce di rete, è `SIOCGIFCONF`; però per ragioni di compatibilità questa operazione è disponibile soltanto per i socket della famiglia `AF_INET` (vale ad dire per socket IPv4). In questo caso l'utente dovrà passare come argomento una struttura `ifconf`, definita in fig. 16.25.

```

struct ifconf {
    int                ifc_len; /* size of buffer */
    union {
        char *         ifc_buf; /* buffer address */
        struct ifreq * ifc_req; /* array of structures */
    };
};

```

Figura 16.25: La struttura `ifconf`.

Per eseguire questa operazione occorrerà allocare preventivamente un buffer di contenente un vettore di strutture `ifreq`. La dimensione (in byte) di questo buffer deve essere specificata nel campo `ifc_len` di `ifconf`, mentre il suo indirizzo andrà specificato nel campo `ifc_req`. Qualora il buffer sia stato allocato come una stringa, il suo indirizzo potrà essere fornito usando il campo `ifc_buf`.⁵²

La funzione restituisce nel buffer indicato una serie di strutture `ifreq` contenenti nel campo `ifr_name` il nome dell'interfaccia e nel campo `ifr_addr` il relativo indirizzo IP. Se lo spazio allocato nel buffer è sufficiente il kernel scriverà una struttura `ifreq` per ciascuna interfaccia attiva, restituendo nel campo `ifc_len` il totale dei byte effettivamente scritti. Il valore di ritorno è 0 se l'operazione ha avuto successo e negativo in caso contrario.

Si tenga presente che il kernel non scriverà mai sul buffer di uscita dati eccedenti numero di byte specificato col valore di `ifc_len` impostato alla chiamata della funzione, troncando il risultato se questi non dovessero essere sufficienti. Questa condizione non viene segnalata come errore per cui occorre controllare il valore di `ifc_len` all'uscita della funzione, e verificare che esso sia inferiore a quello di ingresso. In caso contrario si è probabilmente⁵³ avuta una situazione di troncamento dei dati.

Come esempio dell'uso di queste funzioni si è riportato in fig. 16.26 il corpo principale del programma `iflist` in cui si utilizza l'operazione `SIOCGIFCONF` per ottenere una lista delle interfacce attive e dei relativi indirizzi. Al solito il codice completo è fornito nei sorgenti allegati alla guida.

Il programma inizia (7-11) con la creazione del socket necessario ad eseguire l'operazione, dopo di che si inizializzano opportunamente (13-14) i valori della struttura `ifconf` indicando la dimensione del buffer ed il suo indirizzo;⁵⁴ si esegue poi l'operazione invocando `ioctl`, controllando come sempre la corretta esecuzione, ed uscendo in caso di errore (15-19).

Si esegue poi un controllo sulla quantità di dati restituiti segnalando un eventuale overflow del buffer (21-23); se invece è tutto a posto (24-27) si calcola e si stampa a video il numero di interfacce attive trovate. L'ultima parte del programma (28-33) è il ciclo sul contenuto delle varie strutture `ifreq` restituite in cui si estrae (30) l'indirizzo ad esse assegnato⁵⁵ e lo si stampa (31-32) insieme al nome dell'interfaccia.

16.3.3 L'uso di `ioctl` per i socket TCP e UDP

Non esistono operazioni specifiche per i socket IP in quanto tali,⁵⁶ mentre per i pacchetti di altri protocolli trasportati su IP, qualora li si gestisca attraverso dei socket, si dovrà fare riferimento direttamente all'eventuale supporto presente per il tipo di socket usato: ad esempio si possono ricevere pacchetti ICMP con socket di tipo `raw`, nel qual caso si dovrà fare riferimento alle operazioni di quest'ultimo.

Tuttavia la gran parte dei socket utilizzati nella programmazione di rete utilizza proprio il protocollo IP, e quello che succede è che in realtà la funzione `ioctl` consente di effettuare alcune operazioni specifiche per i socket che usano questo protocollo, ma queste vendono

⁵²si noti che l'indirizzo del buffer è definito in `ifconf` con una union, questo consente di utilizzare una delle due forme a piacere.

⁵³probabilmente perché si potrebbe essere nella condizione in cui sono stati usati esattamente quel numero di byte.

⁵⁴si noti come in questo caso si sia specificato l'indirizzo usando il campo `ifc_buf`, mentre nel seguito del programma si accederà ai valori contenuti nel buffer usando `ifc_req`.

⁵⁵si è definito `access` come puntatore ad una struttura di tipo `sockaddr_in` per poter eseguire un *casting* dell'indirizzo del valore restituito nei vari campi `ifr_addr`, così poi da poterlo poi usare come argomento di `inet_ntoa`.

⁵⁶a parte forse `SIOCGIFCONF`, che però resta attinente alle proprietà delle interfacce di rete, per cui l'abbiamo trattata in sez. 16.3.2 insieme alle altre che comunque si applicano anche ai socket IP.

```

1 int i, num, ret, sock;
2 struct ifconf iflist;
3 char buffer[4096];
4 struct sockaddr_in * address;
5 ...
6 /* create a socket for the operation */
7 sock = socket(PF_INET, SOCK_STREAM, 0);
8 if (sock < 0) {
9     perror("Socket_creation_error");
10    return 1;
11 }
12 /* init values for the ifcon structure and do SIOCGIFCONF */
13 iflist.ifc_len = sizeof(buffer);
14 iflist.ifc_buf = buffer;
15 ret = ioctl(sock, SIOCGIFCONF, &iflist);
16 if (ret < 0) {
17     perror("ioctl_failed");
18     return 1;
19 }
20 /* check that we have all data */
21 if (iflist.ifc_len == sizeof(buffer)) {
22     printf("Probable_overflow,_too_many_interfaces,_cannot_read\n");
23     return 1;
24 } else {
25     num = iflist.ifc_len/sizeof(struct ifreq);
26     printf("Found_%i_interfaces_\n", num);
27 }
28 /* loop on interface to write data */
29 for (i=0; i < num; i++) {
30     address = (struct sockaddr_in *) &iflist.ifc_req[i].ifr_addr;
31     printf("Interface_%s,_address_%s\n", iflist.ifc_req[i].ifr_name,
32           inet_ntoa(address->sin_addr));
33 }
34 return 0;

```

Figura 16.26: Il corpo principale del programma iflist.c.

eseguite, invece che a livello di IP, al successivo livello di trasporto, vale a dire in maniera specifica per i socket TCP e UDP.

Le operazioni di controllo disponibili per i socket TCP sono illustrate dalla relativa pagina di manuale, accessibile con `man 7 tcp`, e prevedono come possibile valore per il secondo argomento della funzione le costanti illustrate nell'elenco seguente; il terzo argomento della funzione, gestito come *value result argument*, deve essere sempre il puntatore ad una variabile di tipo `int`:

SIOCIHQ restituisce la quantità di dati non ancora letti presenti nel buffer di ricezione; il socket non deve essere in stato `LISTEN`, altrimenti si avrà un errore di `EINVAL`.

SIOCATMARK

ritorna un intero non nullo, da intendere come valore logico, se il flusso di dati letti sul socket è arrivato sulla posizione (detta anche *urgent mark*) in cui sono stati ricevuti dati urgenti (vedi sez. 18.1.4). Una operazione di lettura da un socket non attraversa mai questa posizione, per cui è possibile controllare se la si è raggiunta o meno con questa operazione.

Questo è utile quando si attiva l'opzione `SO_OOBINLINE` (vedi sez. 16.2.2) per ricevere i dati urgenti all'interno del flusso dei dati ordinari del socket;⁵⁷ in tal caso quando **SIOCATMARK** restituisce un valore non nullo si saprà che la successiva lettura dal socket restituirà i dati urgenti e non il normale traffico; torneremo su questo in maggior dettaglio in sez. 18.1.4.

SIOCOUTQ restituisce la quantità di dati non ancora inviati presenti nel buffer di spedizione; come per **SIOCIHQ** il socket non deve essere in stato `LISTEN`, altrimenti si avrà un errore di `EINVAL`.

Le operazioni di controllo disponibili per i socket UDP, anch'esse illustrate dalla relativa pagina di manuale accessibile con `man 7 udp`, sono quelle indicate dalle costanti del seguente elenco; come per i socket TCP il terzo argomento viene gestito come *value result argument* e deve essere un puntatore ad una variabile di tipo `int`:

FIONREAD restituisce la dimensione in byte del primo pacchetto in attesa di ricezione, o 0 qualora non ci sia nessun pacchetto.

TIOCOUTQ restituisce il numero di byte presenti nella coda di invio locale; questa opzione è supportata soltanto a partire dal kernel 2.4

16.4 La gestione con `sysctl` ed il filesystem `/proc`

Come ultimo argomento di questo capitolo tratteremo l'uso della funzione `sysctl` (che è stata introdotta nelle sue funzionalità generiche in sez. 6.1.3) per quanto riguarda le sue capacità di effettuare impostazioni relative alle proprietà dei socket. Dato che le stesse funzionalità sono controllabili direttamente attraverso il filesystem `/proc`, le tratteremo attraverso i file presenti in quest'ultimo.

⁵⁷vedremo in sez. 18.1.4 che in genere i dati urgenti presenti su un socket si leggono *out-of-band* usando un opportuno flag per `recvmsg`.

16.4.1 L'uso di `sysctl` e `/proc` per le proprietà della rete

La differenza nell'uso di `sysctl` e del filesystem `/proc` rispetto a quello delle funzioni `ioctl` e `fcntl` visto in sez. 16.3 o all'uso di `getsockopt` e `setsockopt` è che queste funzioni consentono di controllare le proprietà di un singolo socket, mentre con `sysctl` e con `/proc` si impostano proprietà (o valori di default) validi a livello dell'intero sistema, e cioè per tutti i socket.

Le opzioni disponibili per le proprietà della rete, nella gerarchia dei valori impostabili con `sysctl`, sono riportate sotto il nodo `net`, o, se acceduti tramite l'interfaccia del filesystem `/proc`, sotto `/proc/sys/net`. In genere sotto questa directory compaiono le sottodirectory (corrispondenti ad altrettanti sotto-nodi per `sysctl`) relative ai vari protocolli e tipi di interfacce su cui è possibile intervenire per effettuare impostazioni; un contenuto tipico di questa directory è il seguente:

```
/proc/sys/net/
|-- core
|-- ethernet
|-- ipv4
|-- ipv6
|-- irda
|-- token-ring
`-- unix
```

e sono presenti varie centinaia di parametri, molti dei quali non sono neanche documentati; nel nostro caso ci limiteremo ad illustrare quelli più significativi.

Si tenga presente infine che se è sempre possibile utilizzare il filesystem `/proc` come sostituto di `sysctl`, dato che i valori di nodi e sotto-nodi di quest'ultima sono mappati come file e directory sotto `/proc/sys/`, non è vero il contrario, ed in particolare Linux consente di impostare alcuni parametri o leggere lo stato della rete a livello di sistema sotto `/proc/net`, dove sono presenti dei file che non corrispondono a nessun nodo di `sysctl`.

16.4.2 I valori di controllo per i socket generici

Nella directory `/proc/sys/net/core/` sono presenti i file corrispondenti ai parametri generici di `sysctl` validi per tutti i socket. Quelli descritti anche nella pagina di manuale, accessibile con `man 7 socket` sono i seguenti:

<code>rmem_default</code>	imposta la dimensione di default del buffer di ricezione (cioè per i dati in ingresso) dei socket.
<code>rmem_max</code>	imposta la dimensione massima che si può assegnare al buffer di ricezione dei socket attraverso l'uso dell'opzione <code>SO_RCVBUF</code> .
<code>wmem_default</code>	imposta la dimensione di default del buffer di trasmissione (cioè per i dati in uscita) dei socket.
<code>wmem_max</code>	imposta la dimensione massima che si può assegnare al buffer di trasmissione dei socket attraverso l'uso dell'opzione <code>SO_SNDBUF</code> .
<code>message_cost</code> , <code>message_burst</code>	contengono le impostazioni del <i>bucket filter</i> che controlla l'emissione di messaggi di avviso da parte del kernel per eventi relativi a problemi sulla rete,

imponendo un limite che consente di prevenire eventuali attacchi di *Denial of Service* usando i log.⁵⁸

Il *bucket filter* è un algoritmo generico che permette di impostare dei limiti di flusso su una quantità⁵⁹ senza dovere eseguire medie temporali, che verrebbero a dipendere in misura non controllabile dalla dimensione dell'intervallo su cui si media e dalla distribuzione degli eventi;⁶⁰ in questo caso si definisce la dimensione di un “*bidone*” (il *bucket*) e del flusso che da esso può uscire, la presenza di una dimensione iniziale consente di assorbire eventuali picchi di emissione, l’aver fissato un flusso di uscita garantisce che a regime questo sarà il valore medio del flusso ottenibile dal *bucket*.

I due valori indicano rispettivamente il flusso a regime (non sarà inviato più di un messaggio per il numero di secondi specificato da `message_cost`) e la dimensione iniziale per in caso di picco di emissione (verranno accettati inizialmente fino ad un massimo di `message_cost/message_burst` messaggi).

`netdev_max_backlog`

numero massimo di pacchetti che possono essere contenuti nella coda di ingresso generale.

`optmem_max` lunghezza massima dei dati ancillari e di controllo (vedi sez. 18.1.3).

Oltre a questi, nella directory `/proc/sys/net/core` si trovano diversi altri file, la cui documentazione, come per gli altri, dovrebbe essere mantenuta nei sorgenti del kernel nel file `Documentation/networking/ip-sysctl.txt`; la maggior parte di questi però non è documentato:

`dev_weight` blocco di lavoro (*work quantum*) dello *scheduler* di processo dei pacchetti.

`lo_cong` valore per l’occupazione della coda di ricezione sotto la quale si considera di avere una bassa congestione.

`mod_cong` valore per l’occupazione della coda di ricezione sotto la quale si considera di avere una congestione moderata.

`no_cong` valore per l’occupazione della coda di ricezione sotto la quale si considera di non avere congestione.

`no_cong_thresh`

valore minimo (*low water mark*) per il riavvio dei dispositivi congestionati.

`somaxconn` imposta la dimensione massima utilizzabile per il *backlog* della funzione `listen` (vedi sez. 15.2.3), e corrisponde al valore della costante `SOMAXCONN`; il suo valore di default è 128.

⁵⁸senza questo limite un attaccante potrebbe inviare ad arte un traffico che generi intenzionalmente messaggi di errore, per saturare il sistema dei log.

⁵⁹uno analogo viene usato per imporre dei limiti sul flusso dei pacchetti nel *netfilter* di Linux.

⁶⁰in caso di un picco di flusso (il cosiddetto *burst*) il flusso medio verrebbe a dipendere in maniera esclusiva dalla dimensione dell’intervallo di tempo su cui calcola la media.

16.4.3 I valori di controllo per il protocollo IPv4

Nella directory `/proc/sys/net/ipv4` sono presenti i file che corrispondono ai parametri dei socket che usano il protocollo IPv4, relativi quindi sia alle caratteristiche di IP, che a quelle degli altri protocolli che vengono usati all'interno di quest'ultimo (come ICMP, TCP e UDP) o a fianco dello stesso (come ARP).

I file che consentono di controllare le caratteristiche specifiche del protocollo IP in quanto tale, che sono descritti anche nella relativa pagina di manuale accessibile con `man 7 ip`, sono i seguenti:

`ip_default_ttl`

imposta il valore di default per il campo TTL (vedi sez. B.1.2) di tutti i pacchetti uscenti, stabilendo così il numero massimo di router che i pacchetti possono attraversare. Il valore può essere modificato anche per il singolo socket con l'opzione `IP_TTL`. Prende un valore intero, ma dato che il campo citato è di 8 bit hanno senso solo valori fra 0 e 255. Il valore di default è 64, e normalmente non c'è nessuna necessità di modificarlo.⁶¹ Aumentare il valore è una pratica poco gentile, in quanto in caso di problemi di routing si allunga inutilmente il numero di ritrasmissioni.

`ip_forward`

abilita l'inoltro dei pacchetti da una interfaccia ad un'altra, e può essere impostato anche per la singola interfaccia. Prende un valore logico (0 disabilita, diverso da zero abilita), di default è disabilitato.

`ip_dynaddr`

abilita la riscrittura automatica degli indirizzi associati ad un socket quando una interfaccia cambia indirizzo. Viene usato per le interfacce usate nei collegamenti in dial-up, il cui indirizzo IP viene assegnato dinamicamente dal provider, e può essere modificato. Prende un valore intero, con 0 si disabilita la funzionalità, con 1 la si abilita, con 2 (o con qualunque altro valore diverso dai precedenti) la si abilita in modalità *prolissa*; di default la funzionalità è disabilitata.

`ip_autoconfig`

specifica se l'indirizzo IP è stato configurato automaticamente dal kernel all'avvio attraverso DHCP, BOOTP o RARP. Riporta un valore logico (0 falso, 1 vero) accessibile solo in lettura, è inutilizzato nei kernel recenti ed eliminato a partire dal kernel 2.6.18.

`ip_local_port_range`

imposta l'intervallo dei valori usati per l'assegnazione delle porte effimere, permette cioè di modificare i valori illustrati in fig. 15.4; prende due valori interi separati da spazi, che indicano gli estremi dell'intervallo. Si abbia cura di non definire un intervallo che si sovrappone a quello delle porte usate per il *masquerading*, il kernel può gestire la sovrapposizione, ma si avrà una perdita di prestazioni. Si imposti sempre un valore iniziale maggiore di 1024 (o meglio ancora di 4096) per evitare conflitti con le porte usate dai servizi noti.

`ip_no_pmtu_disc`

permette di disabilitare per i socket `SOCK_STREAM` la ricerca automatica della *Path MTU* (vedi sez. 13.3.5 e sez. 16.2.4). Prende un valore logico, e di default è disabilitato (cioè la ricerca viene eseguita).

⁶¹l'unico motivo sarebbe per raggiungere macchine estremamente "lontane" in termini di *hop*, ma è praticamente impossibile trovarne.

In genere si abilita questo parametro quando per qualche motivo il procedimento del *Path MTU discovery* fallisce; dato che questo può avvenire a causa di router⁶² o interfacce⁶³ mal configurati è opportuno correggere le configurazioni, perché disabilitare globalmente il procedimento con questo parametro ha pesanti ripercussioni in termini di prestazioni di rete.

`ip_always_defrag`

fa sì che tutti i pacchetti IP frammentati siano riassemblati, anche in caso di successivo immediato inoltro.⁶⁴ Prende un valore logico e di default è disabilitato. Con i kernel dalla serie 2.4 in poi la deframmentazione viene attivata automaticamente quando si utilizza il sistema del *netfilter*, e questo parametro non è più presente.

`ipfrag_high_thresh`

indica il limite massimo (espresso in numero di byte) sui pacchetti IP frammentati presenti in coda; quando questo valore viene raggiunto la coda viene ripulita fino al valore `ipfrag_low_thresh`. Prende un valore intero.

`ipfrag_low_thresh`

indica la dimensione (specificata in byte) della soglia inferiore a cui viene riportata la coda dei pacchetti IP frammentati quando si raggiunge il valore massimo dato da `ipfrag_high_thresh`. Prende un valore intero.

`ip_nonlocal_bind`

se abilitato rende possibile ad una applicazione eseguire `bind` anche su un indirizzo che non è presente su nessuna interfaccia locale. Prende un valore logico e di default è disabilitato. La funzionalità può essere abilitata a livello di singolo socket con l'opzione `IP_FREEBIND` illustrata in sez. 16.2.4.

Questo può risultare utile per applicazioni particolari (come gli *sniffer*) che hanno la necessità di ricevere pacchetti anche non diretti agli indirizzi presenti sulla macchina, ad esempio per intercettare il traffico per uno specifico indirizzo che si vuole tenere sotto controllo. Il suo uso però può creare problemi ad alcune applicazioni.

I file di `/proc/sys/net/ipv4` che invece fanno riferimento alle caratteristiche specifiche del protocollo TCP, elencati anche nella rispettiva pagina di manuale (accessibile con `man 7 tcp`), sono i seguenti:

`tcp_abort_on_overflow`

indica al kernel di azzerare le connessioni quando il programma che le riceve è troppo lento ed incapace di accettarle. Prende un valore logico ed è disabilitato di default. Questo consente di recuperare le connessioni se si è avuto un eccesso dovuto ad un qualche picco di traffico, ma ovviamente va a discapito dei client che interrogano il server. Pertanto è da abilitare soltanto quando si è sicuri che non è possibile ottimizzare il server in modo che sia in grado di accettare connessioni più rapidamente.

⁶²ad esempio se si scartano tutti i pacchetti ICMP, il problema è affrontato anche in sez. 3.4.4 di [?].

⁶³ad esempio se i due capi di un collegamento *point-to-point* non si accordano sulla stessa MTU.

⁶⁴introdotta con il kernel 2.2.13, nelle versioni precedenti questo comportamento poteva essere solo stabilito una volta per tutte in fase di compilazione del kernel con l'opzione `CONFIG_IP_ALWAYS_DEFRAG`.

tcp_adv_win_scale

indica al kernel quale frazione del buffer associato ad un socket⁶⁵ deve essere utilizzata per la finestra del protocollo TCP⁶⁶ e quale come buffer applicativo per isolare la rete dalle latenze dell'applicazione. Prende un valore intero che determina la suddetta frazione secondo la formula $\text{buffer}/2^{\text{tcp_adv_win_scale}}$ se positivo o con $\text{buffer} - \text{buffer}/2^{\text{tcp_adv_win_scale}}$ se negativo. Il default è 2 che significa che al buffer dell'applicazione viene riservato un quarto del totale.

tcp_app_win

indica la frazione della finestra TCP che viene riservata per gestire l'overhead dovuto alla bufferizzazione. Prende un valore intero che consente di calcolare la dimensione in byte come il massimo fra la MSS e $\text{window}/2^{\text{tcp_app_win}}$. Un valore nullo significa che non viene riservato nessuno spazio; il valore di default è 31.

tcp_dsack

abilita il supporto, definito nell'RFC 2884, per il cosiddetto *Duplicate SACK*.⁶⁷ Prende un valore logico e di default è abilitato.

tcp_ecn

abilita il meccanismo della *Explicit Congestion Notification* (in breve ECN) nelle connessioni TCP. Prende valore logico che di default è disabilitato. La *Explicit Congestion Notification* è un meccanismo che consente di notificare quando una rotta o una rete è congestionata da un eccesso di traffico,⁶⁸ si può così essere avvisati e cercare rotte alternative oppure diminuire l'emissione di pacchetti (in modo da non aumentare la congestione).

Si tenga presente che se si abilita questa opzione si possono avere dei malfunzionamenti apparentemente casuali dipendenti dalla destinazione, dovuti al fatto che alcuni vecchi router non supportano il meccanismo ed alla sua attivazione scartano i relativi pacchetti, bloccando completamente il traffico.

tcp_fack

abilita il supporto per il *TCP Forward Acknowledgement*, un algoritmo per il controllo della congestione del traffico. Prende un valore logico e di default è abilitato.

tcp_fin_timeout

specifica il numero di secondi da passare in stato `FIN_WAIT2` nell'attesa delle ricezione del pacchetto `FIN` conclusivo, passati quali il socket viene comunque chiuso forzatamente. Prende un valore intero che indica i secondi e di default è 60.⁶⁹ L'uso di questa opzione realizza quella che in sostanza è una violazione delle specifiche del protocollo TCP, ma è utile per fronteggiare alcuni attacchi di *Denial of Service*.

tcp_frto

abilita il supporto per l'algoritmo `F-RTO`, un algoritmo usato per la ritrasmissione dei timeout del protocollo TCP, che diventa molto utile per le reti wireless dove la perdita di pacchetti è usualmente dovuta a delle interferenze radio, piuttosto che alla congestione dei router. Prende un valore logico e di default è disabilitato.

⁶⁵quello impostato con `tcp_rmem`.

⁶⁶in sostanza il valore che costituisce la *advertised window* annunciata all'altro capo del socket.

⁶⁷si indica con *SACK (Selective Acknowledgement)* un'opzione TCP, definita nell'RFC 2018, usata per dare un *acknowledgement* unico su blocchi di pacchetti non contigui, che consente di diminuire il numero di pacchetti scambiati.

⁶⁸il meccanismo è descritto in dettaglio nell'RFC 3168 mentre gli effetti sulle prestazioni del suo utilizzo sono documentate nell'RFC 2884.

⁶⁹nei kernel della serie 2.2.x era il valore utilizzato era invece di 120 secondi.

tcp_keepalive_intvl

indica il numero di secondi che deve trascorrere fra l'emissione di due successivi pacchetti di test quando è abilitata la funzionalità del *keepalive* (vedi sez. 16.2.3). Prende un valore intero che di default è 75.

tcp_keepalive_probes

indica il massimo numero pacchetti di *keepalive* (vedi sez. 16.2.3) che devono essere inviati senza ricevere risposta prima che il kernel decida che la connessione è caduta e la termini. Prende un valore intero che di default è 9.

tcp_keepalive_time

indica il numero di secondi che devono passare senza traffico sulla connessione prima che il kernel inizi ad inviare pacchetti di *keepalive*.⁷⁰ Prende un valore intero che di default è 7200, pari a due ore.

tcp_low_latency

indica allo stack TCP del kernel di ottimizzare il comportamento per ottenere tempi di latenza più bassi a scapito di valori più alti per l'utilizzo della banda. Prende un valore logico che di default è disabilitato in quanto un maggior utilizzo della banda è preferito, ma esistono applicazioni particolari in cui la riduzione della latenza è più importante (ad esempio per i cluster di calcolo parallelo) nelle quali lo si può abilitare.

tcp_max_orphans

indica il numero massimo di socket TCP “orfani” (vale a dire non associati a nessun file descriptor) consentito nel sistema.⁷¹ Quando il limite viene ecceduto la connessione orfana viene resettata e viene stampato un avvertimento. Questo limite viene usato per contrastare alcuni elementari attacchi di *denial of service*. Diminuire il valore non è mai raccomandato, in certe condizioni di rete può essere opportuno aumentarlo, ma si deve tenere conto del fatto che ciascuna connessione orfana può consumare fino a 64K di memoria del kernel. Prende un valore intero, il valore di default viene impostato inizialmente al valore del parametro del kernel NR_FILE, e viene aggiustato a seconda della memoria disponibile.

tcp_max_syn_backlog

indica la lunghezza della coda delle connessioni incomplete, cioè delle connessioni per le quali si è ricevuto un SYN di richiesta ma non l'ACK finale del *three way handshake* (si riveda quanto illustrato in sez. 15.2.3).

Quando questo valore è superato il kernel scarterà immediatamente ogni ulteriore richiesta di connessione. Prende un valore intero; il default, che è 256, viene automaticamente portato a 1024 qualora nel sistema ci sia sufficiente memoria (se maggiore di 128Mb) e ridotto a 128 qualora la memoria sia poca (inferiore a 32Mb).⁷²

tcp_max_tw_buckets

indica il numero massimo di socket in stato TIME_WAIT consentito nel sistema. Prende un valore intero di default è impostato al doppio del valore del parametro NR_FILE, ma che viene aggiustato automaticamente a seconda della

⁷⁰ha effetto solo per i socket per cui si è impostata l'opzione SO_KEEPALIVE (vedi sez. 16.2.3).

⁷¹trattasi in genere delle connessioni relative a socket chiusi che non hanno completato il processo di chiusura.

⁷²si raccomanda, qualora si voglia aumentare il valore oltre 1024, di seguire la procedura citata nella pagina di manuale di TCP, e modificare il valore della costante TCP_SYNQ_HSIZE nel file include/net/tcp.h dei sorgenti del kernel, in modo che sia $\text{tcp_max_syn_backlog} \geq 16 * \text{TCP_SYNQ_HSIZE}$, per poi ricompilare il kernel.

memoria presente. Se il valore viene superato il socket viene chiuso con la stampa di un avviso; l'uso di questa funzionalità consente di prevenire alcuni semplici attacchi di *denial of service*.

tcp_mem viene usato dallo stack TCP per gestire le modalità con cui esso utilizzerà la memoria. Prende una tripletta di valori interi, che indicano un numero di pagine:

- il primo valore, chiamato *low* nelle pagine di manuale, indica il numero di pagine allocate sotto il quale non viene usato nessun meccanismo di regolazione dell'uso della memoria.
- il secondo valore, chiamato *pressure* indica il numero di pagine allocate passato il quale lo stack TCP inizia a moderare il suo consumo di memoria; si esce da questo stato di *pressione* sulla memoria quando il numero di pagine scende sotto il precedente valore *low*.
- il terzo valore, chiamato *high* indica il numero massimo di pagine che possono essere utilizzate dallo stack TCP/IP, e soprassiede ogni altro valore specificato dagli altri limiti del kernel.

tcp_orphan_retries

indica il numero massimo di volte che si esegue un tentativo di controllo sull'altro capo di una connessione che è stata già chiusa dalla nostra parte. Prende un valore intero che di default è 8.

tcp_reordering

indica il numero massimo di volte che un pacchetto può essere riordinato nel flusso di dati, prima che lo stack TCP assuma che è andato perso e si ponga nello stato di *slow start* (si veda sez. ??) viene usata questa metrica di riconoscimento dei riordinamenti per evitare inutili ritrasmissioni provocate dal riordinamento. Prende un valore intero che di default che è 3, e che non è opportuno modificare.

tcp_retrans_collapse

in caso di pacchetti persi durante una connessione, per ottimizzare l'uso della banda il kernel cerca di eseguire la ritrasmissione inviando pacchetti della massima dimensione possibile; in sostanza dati che in precedenza erano stati trasmessi su pacchetti diversi possono essere ritrasmessi riuniti su un solo pacchetto (o su un numero minore di pacchetti di dimensione maggiore). Prende un valore logico e di default è abilitato.

tcp_retries1 imposta il massimo numero di volte che protocollo tenterà la ritrasmissione di un pacchetto su una connessione stabilita prima di fare ricorso ad ulteriori sforzi che coinvolgano anche il livello di rete. Passato questo numero di ritrasmissioni verrà fatto eseguire al livello di rete un tentativo di aggiornamento della rotta verso la destinazione prima di eseguire ogni successiva ritrasmissione. Prende un valore intero che di default è 3.

tcp_retries2 imposta il numero di tentativi di ritrasmissione di un pacchetto inviato su una connessione già stabilita per il quale non si sia ricevuto una risposta di ACK (si veda anche quanto illustrato in sez. 15.5.2). Prende un valore intero che di default è 15, il che comporta un tempo variabile fra 13 e 30 minuti; questo non corrisponde a quanto richiesto nell'RFC 1122 dove è indicato un massimo di 100 secondi, che però è un valore considerato troppo basso.

- tcp_rfc1337** indica al kernel di abilitare il comportamento richiesto nell’RFC 1337. Prende un valore logico e di default è disabilitato, il che significa che alla ricezione di un segmento RST in stato `TIME_WAIT` il socket viene chiuso immediatamente senza attendere la conclusione del periodo di `TIME_WAIT`.
- tcp_rmem** viene usato dallo stack TCP per controllare dinamicamente le dimensioni dei propri buffer di ricezione, anche in rapporto alla memoria disponibile. Prende una tripletta di valori interi separati da spazi che indicano delle dimensioni in byte:
- il primo valore, chiamato *min* nelle pagine di manuale, indica la dimensione minima in byte del buffer di ricezione; il default è 4Kb, ma in sistemi con poca memoria viene automaticamente ridotto a `PAGE_SIZE`. Questo valore viene usato per assicurare che anche in situazioni di pressione sulla memoria (vedi quanto detto per `tcp_rmem`) le allocazioni al di sotto di questo limite abbiamo comunque successo. Questo valore non viene comunque ad incidere sulla dimensione del buffer di ricezione di un singolo socket dichiarata con l’opzione `SO_RCVBUF`.
 - il secondo valore, denominato *default* nelle pagine di manuale, indica la dimensione di default, in byte, del buffer di ricezione di un socket TCP. Questo valore sovrascrive il default iniziale impostato per tutti i socket con `/proc/sys/net/core/mem_default` che vale per qualunque protocollo. Il default è 87380 byte, ridotto a 43689 per sistemi con poca memoria. Se si desiderano dimensioni più ampie per tutti i socket si può aumentare questo valore, ma se si vuole che in corrispondenza aumentino anche le dimensioni usate per la finestra TCP si deve abilitare il *TCP window scaling* (di default è abilitato, vedi più avanti `tcp_window_scaling`).
 - il terzo valore, denominato *max* nelle pagine di manuale, indica la dimensione massima in byte del buffer di ricezione di un socket TCP; il default è 174760 byte, che viene ridotto automaticamente a 87380 per sistemi con poca memoria. Il valore non può comunque eccedere il limite generale per tutti i socket posto con `/proc/sys/net/core/rmem_max`. Questo valore non viene ad incidere sulla dimensione del buffer di ricezione di un singolo socket dichiarata con l’opzione `SO_RCVBUF`.
- tcp_sack** indica al kernel di utilizzare il meccanismo del *TCP selective acknowledgement* definito nell’RFC 2018. Prende un valore logico e di default è abilitato.
- tcp_stdurg** indica al kernel di utilizzare l’interpretazione che viene data dall’RFC 1122 del puntatore dei *dati urgenti* (vedi sez. 18.1.4) in cui questo punta all’ultimo byte degli stessi; se disabilitato viene usata l’interpretazione usata da BSD per cui esso punta al primo byte successivo. Prende un valore logico e di default è disabilitato, perché abilitarlo può dar luogo a problemi di interoperabilità.
- tcp_synack_retries** indica il numero massimo di volte che verrà ritrasmesso il segmento SYN/ACK nella creazione di una connessione (vedi sez. 15.1.1). Prende un valore intero ed il valore di default è 5; non si deve superare il valore massimo di 255.

tcp_syncookies

abilita i *TCP syncookies*.⁷³ Prende un valore logico, e di default è disabilitato. Questa funzionalità serve a fornire una protezione in caso di un attacco di tipo *SYN flood*, e deve essere utilizzato come ultima risorsa dato che costituisce una violazione del protocollo TCP e configge con altre funzionalità come le estensioni e può causare problemi per i client ed il reinoltro dei pacchetti.

tcp_syn_retries

imposta il numero di tentativi di ritrasmissione dei pacchetti SYN di inizio connessione del *three way handshake* (si ricordi quanto illustrato in sez. 15.2.2). Prende un valore intero che di default è 5; non si deve superare il valore massimo di 255.

tcp_timestamps

abilita l'uso dei *TCP timestamps*, come definiti nell'RFC 1323. Prende un valore logico e di default è abilitato.

tcp_tw_recycle

abilita il riutilizzo rapido dei socket in stato `TIME_WAIT`. Prende un valore logico e di default è disabilitato. Non è opportuno abilitare questa opzione che può causare problemi con il NAT.⁷⁴

tcp_tw_reuse abilita il riutilizzo dello stato `TIME_WAIT` quando questo è sicuro dal punto di vista del protocollo. Prende un valore logico e di default è disabilitato.

tcp_window_scaling

un valore logico, attivo di default, che abilita la funzionalità del *TCP window scaling* definita dall'RFC 1323. Prende un valore logico e di default è abilitato. Come accennato in sez. 15.1.2 i 16 bit della finestra TCP comportano un limite massimo di dimensione di 64Kb, ma esiste una opportuna opzione del protocollo che permette di applicare un fattore di scale che consente di aumentarne le dimensioni. Questa è pienamente supportata dallo stack TCP di Linux, ma se lo si disabilita la negoziazione del *TCP window scaling* con l'altro capo della connessione non viene effettuata.

tcp_wmem viene usato dallo stack TCP per controllare dinamicamente le dimensioni dei propri buffer di spedizione, adeguandole in rapporto alla memoria disponibile. Prende una tripletta di valori interi separati da spazi che indicano delle dimensioni in byte:

- il primo valore, chiamato *min*, indica la dimensione minima in byte del buffer di spedizione; il default è 4Kb. Come per l'analogo di `tcp_rmem` viene usato per assicurare che anche in situazioni di pressione sulla memoria (vedi `tcp_mem`) le allocazioni al di sotto di questo limite abbiano comunque successo. Di nuovo questo valore non viene ad incidere sulla dimensione del buffer di trasmissione di un singolo socket dichiarata con l'opzione `SO_SNDBUF`.

⁷³per poter usare questa funzionalità è necessario avere abilitato l'opzione `CONFIG_SYN_COOKIES` nella compilazione del kernel.

⁷⁴la *Network Address Translation* (abbreviato in NAT) è una tecnica, impiegata nei firewall e nei router, che consente di modificare al volo gli indirizzi dei pacchetti che transitano per una macchina, Linux la supporta con il *netfilter*.

- il secondo valore, denominato *default*, indica la dimensione di default in byte del buffer di spedizione di un socket TCP. Questo valore sovrascrive il default iniziale impostato per tutti i tipi di socket sul file `/proc/sys/net/core/wmem_default`. Il default è 87380 byte, ridotto a 43689 per sistemi con poca memoria. Si può aumentare questo valore quando si desiderano dimensioni più ampie del buffer di trasmissione per i socket TCP, ma come per il precedente `tcp_rmem` se si vuole che in corrispondenza aumentino anche le dimensioni usate per la finestra TCP si deve abilitare il *TCP window scaling* con `tcp_window_scaling`.
- il terzo valore, denominato *max*, indica la dimensione massima in byte del buffer di spedizione di un socket TCP; il default è 128Kb, che viene ridotto automaticamente a 64Kb per sistemi con poca memoria. Il valore non può comunque eccedere il limite generale per tutti i socket posto con `/proc/sys/net/core/wmem_max`. Questo valore non viene ad incidere sulla dimensione del buffer di trasmissione di un singolo socket dichiarata con l'opzione `SO_SNDBUF`.

Capitolo 17

Gli altri tipi di socket

Dopo aver trattato in cap. 15 i socket TCP, che costituiscono l'esempio più comune dell'interfaccia dei socket, esamineremo in questo capitolo gli altri tipi di socket, a partire dai socket UDP, e i socket *Unix domain* già incontrati in sez. 11.1.5.

17.1 I socket UDP

Dopo i socket TCP i socket più utilizzati nella programmazione di rete sono i socket UDP: protocolli diffusi come NFS o il DNS usano principalmente questo tipo di socket. Tratteremo in questa sezione le loro caratteristiche principali e le modalità per il loro utilizzo.

17.1.1 Le caratteristiche di un socket UDP

Come illustrato in sez.13.3.3 UDP è un protocollo molto semplice che non supporta le connessioni e non è affidabile: esso si appoggia direttamente sopra IP (per i dettagli sul protocollo si veda sez. C.2). I dati vengono inviati in forma di pacchetti, e non ne è assicurata né la effettiva ricezione né l'arrivo nell'ordine in cui vengono inviati. Il vantaggio del protocollo è la velocità, non è necessario trasmettere le informazioni di controllo ed il risultato è una trasmissione di dati più veloce ed immediata.

Questo significa che a differenza dei socket TCP i socket UDP non supportano una comunicazione di tipo *stream* in cui si ha a disposizione un flusso continuo di dati che può essere letto un po' alla volta, ma piuttosto una comunicazione di tipo *datagram*, in cui i dati arrivano in singoli blocchi che devono essere letti integralmente.

Questo diverso comportamento significa anche che i socket UDP, pur appartenendo alla famiglia PF_INET¹ devono essere aperti quando si usa la funzione `socket` (si riveda quanto illustrato a suo tempo in tab. 14.2) utilizzando per il tipo di socket il valore `SOCK_DGRAM`.

Questa differenza comporta ovviamente che anche le modalità con cui si usano i socket UDP sono completamente diverse rispetto ai socket TCP, ed in particolare non esistendo il concetto di connessione non esiste il meccanismo del *three way handshake* né quello degli stati del protocollo. In realtà tutto quello che avviene nella comunicazione attraverso dei socket UDP è la trasmissione di un pacchetto da un client ad un server o viceversa, secondo lo schema illustrato in fig. 17.1.

¹o PF_INET6 qualora si usasse invece il protocollo IPv6, che pure supporta UDP.

Figura 17.1: Lo schema di interscambio dei pacchetti per una comunicazione via UDP.

Come illustrato in fig. 17.1 la struttura generica di un server UDP prevede, una volta creato il socket, la chiamata a `bind` per mettersi in ascolto dei dati, questa è l'unica parte comune con un server TCP. Non essendovi il concetto di connessione le funzioni `listen` ed `accept` non sono mai utilizzate nel caso di server UDP. La ricezione dei dati dal client avviene attraverso la funzione `recvfrom`, mentre una eventuale risposta sarà inviata con la funzione `sendto`.

Da parte del client invece, una volta creato il socket non sarà necessario connettersi con `connect` (anche se, come vedremo in sez. 17.1.6, è possibile usare questa funzione, con un significato comunque diverso) ma si potrà effettuare direttamente una richiesta inviando un pacchetto con la funzione `sendto` e si potrà leggere una eventuale risposta con la funzione `recvfrom`.

Anche se UDP è completamente diverso rispetto a TCP resta identica la possibilità di gestire più canali di comunicazione fra due macchine utilizzando le porte. In questo caso il server dovrà usare comunque la funzione `bind` per scegliere la porta su cui ricevere i dati, e come nel caso dei socket TCP si potrà usare il comando `netstat` per verificare quali socket sono in ascolto:

```
[piccardi@gont gapil]# netstat -anu
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
udp      0      0 0.0.0.0:32768           0.0.0.0:*
udp      0      0 192.168.1.2:53          0.0.0.0:*
udp      0      0 127.0.0.1:53            0.0.0.0:*
udp      0      0 0.0.0.0:67              0.0.0.0:*
```

in questo caso abbiamo attivi il DNS (sulla porta 53, e sulla 32768 per la connessione di controllo del server `named`) ed un server DHCP (sulla porta 67).

Si noti però come in questo caso la colonna che indica lo stato sia vuota. I socket UDP infatti non hanno uno stato. Inoltre anche in presenza di traffico non si avranno indicazioni delle connessioni attive, proprio perché questo concetto non esiste per i socket UDP, il kernel si limita infatti a ricevere i pacchetti ed inviarli al processo in ascolto sulla porta cui essi sono destinati, oppure a scartarli inviando un messaggio *ICMP port unreachable* qualora non vi sia nessun processo in ascolto.

17.1.2 Le funzioni `sendto` e `recvfrom`

Come accennato in sez. 17.1.1 le due funzioni principali usate per la trasmissione di dati attraverso i socket UDP (ed in generale per i socket di tipo *datagram*) sono `sendto` e `recvfrom`. La necessità di usare queste funzioni è dovuta al fatto che non esistendo con UDP il concetto di connessione, non si può stabilire (come avviene con i socket TCP grazie alla chiamata ad `accept` che li associa ad una connessione) quali sono la sorgente e la destinazione dei dati che passano sul socket.²

Per questo anche se in generale si possono comunque leggere i dati con `read`, usando questa funzione non si sarà in grado di determinare da quale fra i possibili corrispondenti (se

²anche se in alcuni casi, come quello di un client che contatta un server, è possibile connettere il socket (vedi sez. 17.1.6), la necessità di `sendto` e `recvfrom` resta, dato che questo è possibile solo sul lato del client.

ve ne sono più di uno, come avviene sul lato del server) questi arrivino. E non sarà comunque possibile usare `write` (che fallisce un errore di `EDESTADDRREQ`) in quanto non è determinato la destinazione che i dati avrebbero.

Per questo motivo nel caso di UDP diventa essenziale utilizzare queste due funzioni, che sono comunque utilizzabili in generale per la trasmissione di dati attraverso qualunque tipo di socket. Esse hanno la caratteristica di prevedere tre argomenti aggiuntivi attraverso i quali è possibile specificare la destinazione dei dati trasmessi o ottenere l'origine dei dati ricevuti. La prima di queste funzioni è `sendto` ed il suo prototipo³ è:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *to,
               socklen_t tolen)
    Trasmette un messaggio ad un altro socket.
```

La funzione restituisce il numero di caratteri inviati in caso di successo e -1 per un errore; nel qual caso `errno` viene impostata al rispettivo codice di errore:

`EAGAIN` il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blocchi.

`ECONNRESET` l'altro capo della comunicazione ha resettato la connessione.

`EDESTADDRREQ` il socket non è di tipo connesso, e non si è specificato un indirizzo di destinazione.

`EISCONN` il socket è già connesso, ma si è specificato un destinatario.

`EMSGSIZE` il tipo di socket richiede l'invio dei dati in un blocco unico, ma la dimensione del messaggio lo rende impossibile.

`ENOBUFS` la coda di uscita dell'interfaccia è già piena (di norma Linux non usa questo messaggio ma scarta silenziosamente i pacchetti).

`ENOTCONN` il socket non è connesso e non si è specificata una destinazione.

`EOPNOTSUPP` il valore di flag non è appropriato per il tipo di socket usato.

`EPIPE` il capo locale della connessione è stato chiuso, si riceverà anche un segnale di `SIGPIPE`, a meno di non aver impostato `MSG_NOSIGNAL` in `flags`.

ed anche `EFAULT`, `EBADF`, `EINVAL`, `EINTR`, `ENOMEM`, `ENOSOCK` più gli eventuali altri errori relativi ai protocolli utilizzati.

I primi tre argomenti sono identici a quelli della funzione `write` e specificano il socket `sockfd` a cui si fa riferimento, il buffer `buf` che contiene i dati da inviare e la relativa lunghezza `len`. Come per `write` la funzione ritorna il numero di byte inviati; nel caso di UDP però questo deve sempre corrispondere alla dimensione totale specificata da `len` in quanto i dati vengono sempre inviati in forma di pacchetto e non possono essere spezzati in invii successivi. Qualora non ci sia spazio nel buffer di uscita la funzione si blocca (a meno di non avere aperto il socket in modalità non bloccante), se invece non è possibile inviare il messaggio all'interno di un unico pacchetto (ad esempio perché eccede le dimensioni massime del protocollo sottostante utilizzato) essa fallisce con l'errore di `EMSGSIZE`.

I due argomenti `to` e `tolen` servono a specificare la destinazione del messaggio da inviare, e indicano rispettivamente la struttura contenente l'indirizzo di quest'ultima e la sua dimensione; questi argomenti vanno specificati stessa forma in cui li si sarebbero usati con `connect`. Nel nostro caso `to` dovrà puntare alla struttura contenente l'indirizzo IP e la porta di destinazione verso cui si vogliono inviare i dati (questo è indifferente rispetto all'uso di TCP o UDP, usando socket diversi si sarebbero dovute utilizzare le rispettive strutture degli indirizzi).

³il prototipo illustrato è quello utilizzato dalla *glibc*, che segue le *Single Unix Specification*, l'argomento `flags` era di tipo `int` nei vari BSD4.*, mentre nelle *libc4* e *libc5* veniva usato un `unsigned int`; l'argomento `len` era `int` nei vari BSD4.* e nella *libc4*, ma `size_t` nella *libc5*; infine l'argomento `tolen` era `int` nei vari BSD4.*, nella *libc4* e nella *libc5*.

Se il socket è di un tipo che prevede le connessioni (ad esempio un socket TCP), questo deve essere già connesso prima di poter eseguire la funzione, in caso contrario si riceverà un errore di `ENOTCONN`. In questo specifico caso in cui gli argomenti `to` e `tolen` non servono essi dovranno essere inizializzati rispettivamente a `NULL` e `0`; normalmente quando si opera su un socket connesso essi vengono ignorati, ma qualora si sia specificato un indirizzo è possibile ricevere un errore di `EISCONN`.

Finora abbiamo tralasciato l'argomento `flags`; questo è un intero usato come maschera binaria che permette di impostare una serie di modalità di funzionamento della comunicazione attraverso il socket (come `MSG_NOSIGNAL` che impedisce l'invio del segnale `SIGPIPE` quando si è già chiuso il capo locale della connessione). Torneremo con maggiori dettagli sul significato di questo argomento in sez. 18.1.2, dove tratteremo le funzioni avanzate dei socket, per il momento ci si può limitare ad usare sempre un valore nullo.

La seconda funzione utilizzata nella comunicazione fra socket UDP è `recvfrom`, che serve a ricevere i dati inviati da un altro socket; il suo prototipo⁴ è:

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *from,
socklen_t *fromlen)
    Riceve un messaggio da un socket.
```

La funzione restituisce il numero di byte ricevuti in caso di successo e -1 in caso di errore; nel qual caso `errno` assumerà il valore:

`EAGAIN` il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blocchi, oppure si è impostato un timeout in ricezione e questo è scaduto.

`ECONNREFUSED` l'altro capo della comunicazione ha rifiutato la connessione (in genere perché il relativo servizio non è disponibile).

`ENOTCONN` il socket è di tipo connesso, ma non si è eseguita la connessione.

ed anche `EFAULT`, `EBADF`, `EINVAL`, `EINTR`, `ENOMEM`, `ENOTSOCK` più gli eventuali altri errori relativi ai protocolli utilizzati.

Come per `sendto` i primi tre argomenti sono identici agli analoghi di `read`: dal socket vengono letti `len` byte che vengono salvati nel buffer `buf`. A seconda del tipo di socket (se di tipo *datagram* o di tipo *stream*) i byte in eccesso che non sono stati letti possono rispettivamente andare persi o restare disponibili per una lettura successiva. Se non sono disponibili dati la funzione si blocca, a meno di non aver aperto il socket in modalità non bloccante, nel qual caso si avrà il solito errore di `EAGAIN`. Qualora `len` ecceda la dimensione del pacchetto la funzione legge comunque i dati disponibili, ed il suo valore di ritorno è comunque il numero di byte letti.

I due argomenti `from` e `fromlen` sono utilizzati per ottenere l'indirizzo del mittente del pacchetto che è stato ricevuto, e devono essere opportunamente inizializzati; il primo deve contenere il puntatore alla struttura (di tipo `sockaddr`) che conterrà l'indirizzo e il secondo il puntatore alla variabile con la dimensione di detta struttura. Si tenga presente che mentre il contenuto della struttura `sockaddr` cui punta `from` può essere qualunque, la variabile puntata da `fromlen` deve essere opportunamente inizializzata a `sizeof(sockaddr)`, assicurandosi che la dimensione sia sufficiente a contenere tutti i dati dell'indirizzo.⁵ Al ritorno della funzione si otterranno i dati dell'indirizzo e la sua effettiva lunghezza, (si noti che `fromlen` è un valore

⁴il prototipo è quello della *glibc* che segue le *Single Unix Specification*, i vari BSD4.*, le *libc4* e la *libc5* usano un `int` come valore di ritorno; per gli argomenti `flags` e `len` vale quanto detto a proposito di `sendto`; infine l'argomento `fromlen` è `int` per i vari BSD4.*, la *libc4* e la *libc5*.

⁵si ricordi che `sockaddr` è un tipo generico che serve ad indicare la struttura corrispondente allo specifico tipo di indirizzo richiesto, il valore di `fromlen` pone un limite alla quantità di dati che verranno scritti sulla struttura puntata da `from` e se è insufficiente l'indirizzo risulterà corrotto.

intero ottenuto come *value result argument*). Se non si è interessati a questa informazione, entrambi gli argomenti devono essere inizializzati al valore NULL.

Una differenza fondamentale del comportamento di queste funzioni rispetto alle usuali `read` e `write` che abbiamo usato con i socket TCP è che in questo caso è perfettamente legale inviare con `sendto` un pacchetto vuoto (che nel caso conterrà solo le intestazioni di IP e di UDP), specificando un valore nullo per `len`. Allo stesso modo è possibile ricevere con `recvfrom` un valore di ritorno di 0 byte, senza che questo possa configurarsi come una chiusura della connessione⁶ o come una cessazione delle comunicazioni.

17.1.3 Un client UDP elementare

Vediamo allora come implementare un primo client elementare con dei socket UDP. Ricalcando quanto fatto nel caso dei socket TCP prenderemo come primo esempio l'uso del servizio *daytime*, utilizzando questa volta UDP. Il servizio è definito nell'RFC 867, che nel caso di uso di UDP prescrive che il client debba inviare un pacchetto UDP al server (di contenuto non specificato), il quale risponderà a inviando a sua volta un pacchetto UDP contenente la data.

In fig. 17.2 è riportato la sezione principale del codice del nostro client, il sorgente completo si trova nel file `UDP_daytime.c` distribuito con gli esempi allegati alla guida; al solito si è tralasciato di riportare in figura la sezione relativa alla gestione delle opzioni a riga di comando (nel caso praticamente assenti).

Il programma inizia (9-12) con la creazione del socket, al solito uscendo dopo aver stampato un messaggio in caso errore. Si noti come in questo caso, rispetto all'analogo client basato su socket TCP di fig. 15.8 si sia usato per il tipo di socket il valore `SOCK_DGRAM`, pur mantenendosi nella stessa famiglia data da `AF_INET`.

Il passo successivo (13-21) è l'inizializzazione della struttura degli indirizzi; prima (14) si cancella completamente la stessa con `memset`, (15) poi si imposta la famiglia dell'indirizzo ed infine (16) la porta. Infine (18-21) si ricava l'indirizzo del server da contattare dall'argomento passato a riga di comando, convertendolo con `inet_pton`. Si noti come questa sezione sia identica a quella del client TCP di fig. 15.8, in quanto la determinazione dell'uso di UDP al posto di TCP è stata effettuata quando si è creato il socket.

Una volta completate le inizializzazioni inizia il corpo principale del programma, il primo passo è inviare, come richiesto dal protocollo, un pacchetto al server. Questo lo si fa (16) inviando un pacchetto vuoto (si ricordi quanto detto in sez. 17.1.2) con `sendto`, avendo cura di passare un valore nullo per il puntatore al buffer e la lunghezza del messaggio. In realtà il protocollo non richiede che il pacchetto sia vuoto, ma dato che il server comunque ne ignorerà il contenuto, è inutile inviare dei dati.

Verificato (24-27) che non ci siano stati errori nell'invio si provvede (28) ad invocare `recvfrom` per ricevere la risposta del server. Si controlla poi (29-32) che non vi siano stati errori in ricezione (uscendo con un messaggio in caso contrario); se è tutto a posto la variabile `nread` conterrà la dimensione del messaggio di risposta inviato dal server che è stato memorizzato su `buffer`, se (34) pertanto il valore è positivo si provvederà (35) a terminare la stringa contenuta nel buffer di lettura⁷ e a stamparla (36) sullo standard output, controllando anche in questo caso (36-38) l'esito dell'operazione, ed uscendo con un messaggio in caso di errore.

⁶dato che la connessione non esiste, non ha senso parlare di chiusura della connessione, questo significa anche che con i socket UDP non è necessario usare `close` o `shutdown` per terminare la comunicazione.

⁷si ricordi che, come illustrato in sez. 15.3.2, il server invia in risposta una stringa contenente la data, terminata dai due caratteri CR e LF, che pertanto prima di essere stampata deve essere opportunamente terminata con un NUL.

```

1 int main(int argc, char *argv[])
2 {
3     int sock;
4     int i, nread;
5     struct sockaddr_in addr;
6     char buffer[MAXLINE];
7     ...
8     /* create socket */
9     if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
10         perror("Socket_creation_error");
11         return -1;
12     }
13     /* initialize address */
14     memset((void *) &addr, 0, sizeof(addr)); /* clear server address */
15     addr.sin_family = AF_INET; /* address type is INET */
16     addr.sin_port = htons(13); /* daytime port is 13 */
17     /* build address using inet_pton */
18     if ( (inet_pton(AF_INET, argv[optind], &addr.sin_addr)) <= 0) {
19         perror("Address_creation_error");
20         return -1;
21     }
22     /* send request packet */
23     nread = sendto(sock, NULL, 0, 0, (struct sockaddr *)&addr, sizeof(addr));
24     if (nread < 0) {
25         perror("Request_error");
26         return -1;
27     }
28     nread = recvfrom(sock, buffer, MAXLINE, 0, NULL, NULL);
29     if (nread < 0) {
30         perror("Read_error");
31         return -1;
32     }
33     /* print results */
34     if (nread > 0) {
35         buffer[nread]=0;
36         if (fputs(buffer, stdout) == EOF) { /* write daytime */
37             perror("fputs_error");
38             return -1;
39         }
40     }
41     /* normal exit */
42     return 0;
43 }

```

Figura 17.2: Sezione principale del client per il servizio *daytime* su UDP.

Se pertanto si è avuto cura di attivare il server del servizio *daytime*⁸ potremo verificare il funzionamento del nostro client interrogando quest'ultimo con:

```
[piccardi@gont sources]$ ./daytime 127.0.0.1
Sat Mar 20 23:17:13 2004
```

ed osservando il traffico con uno sniffer potremo effettivamente vedere lo scambio dei due pacchetti, quello vuoto di richiesta, e la risposta del server:

```
[root@gont gapil]# tcpdump -i lo
tcpdump: listening on lo
23:41:21.645579 localhost.32780 > localhost.daytime: udp 0 (DF)
23:41:21.645710 localhost.daytime > localhost.32780: udp 26 (DF)
```

Una differenza fondamentale del nostro client è che in questo caso, non disponendo di una connessione, è per lui impossibile riconoscere errori di invio relativi alla rete. La funzione `sendto` infatti riporta solo errori locali, i dati vengono comunque scritti e la funzione ritorna senza errori anche se il server non è raggiungibile o non esiste un server in ascolto sull'indirizzo di destinazione. Questo comporta ad esempio che se si usa il nostro programma interrogando un server inesistente questo resterà perennemente bloccato nella chiamata a `recvfrom`, fin quando non lo interromperemo. Vedremo in sez. 17.1.6 come si può porre rimedio a questa problematica.

17.1.4 Un server UDP elementare

Nella sezione precedente abbiamo visto come scrivere un client elementare per servizio *daytime*, vediamo in questa come deve essere scritto un server. Si ricordi che il compito di quest'ultimo è quello di ricevere un pacchetto di richiesta ed inviare in risposta un pacchetto contenente una stringa con la data corrente.

In fig. 17.3 è riportato la sezione principale del codice del nostro client, il sorgente completo si trova nel file `UDP_daytimed.c` distribuito con gli esempi allegati alla guida; anche in questo caso si è omessa la sezione relativa alla gestione delle opzioni a riga di comando (la sola presente è `-v` che permette di stampare a video l'indirizzo associato ad ogni richiesta).

Anche in questo caso la prima parte del server (9-23) è sostanzialmente identica a quella dell'analogo server per TCP illustrato in fig. 15.10; si inizia (10) con il creare il socket, uscendo con un messaggio in caso di errore (10-13), e di nuovo la sola differenza con il caso precedente è il diverso tipo di socket utilizzato. Dopo di che (14-18) si inizializza la struttura degli indirizzi che poi (20) verrà usata da `bind`; si cancella (15) preventivamente il contenuto, si imposta (16) la famiglia dell'indirizzo, la porta (17) e l'indirizzo (18) su cui si riceveranno i pacchetti. Si noti come in quest'ultimo sia l'indirizzo generico `INADDR_ANY`; questo significa (si ricordi quanto illustrato in sez. 15.2.1) che il server accetterà pacchetti su uno qualunque degli indirizzi presenti sulle interfacce di rete della macchina.

Completata l'inizializzazione tutto quello che resta da fare è eseguire (20-23) la chiamata a `bind`, controllando la presenza di eventuali errori, ed uscendo con un avviso qualora questo fosse il caso. Nel caso di socket UDP questo è tutto quello che serve per consentire al server di ricevere i pacchetti a lui indirizzati, e non è più necessario chiamare successivamente `listen`. In questo caso infatti non esiste il concetto di connessione, e quindi non deve essere predisposta una coda delle connessioni entranti. Nel caso di UDP i pacchetti arrivano al kernel con un certo indirizzo ed una certa porta di destinazione, il kernel controlla se corrispondono ad un socket

⁸di norma questo è un servizio standard fornito dal *superdemon* `inetd`, per cui basta abilitarlo nel file di configurazione di quest'ultimo, avendo cura di predisporre il servizio su UDP.

```

1 int main(int argc, char *argv[])
2 {
3     int sock;
4     int i, n, len, verbose=0;
5     struct sockaddr_in addr;
6     char buffer[MAXLINE];
7     time_t timeval;
8     ...
9     /* create socket */
10    if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
11        perror("Socket_creation_error");
12        exit(-1);
13    }
14    /* initialize address */
15    memset((void *)&addr, 0, sizeof(addr));          /* clear server address */
16    addr.sin_family = AF_INET;                       /* address type is INET */
17    addr.sin_port = htons(13);                       /* daytime port is 13 */
18    addr.sin_addr.s_addr = htonl(INADDR_ANY);        /* connect from anywhere */
19    /* bind socket */
20    if (bind(sock, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
21        perror("bind_error");
22        exit(-1);
23    }
24    /* write daytime to client */
25    while (1) {
26        n = recvfrom(sock, buffer, MAXLINE, 0, (struct sockaddr *)&addr, &len);
27        if (n < 0) {
28            perror("recvfrom_error");
29            exit(-1);
30        }
31        if (verbose) {
32            inet_ntop(AF_INET, &addr.sin_addr, buffer, sizeof(buffer));
33            printf("Request_from_host_%s,_port_%d\n", buffer,
34                ntohs(addr.sin_port));
35        }
36        timeval = time(NULL);
37        snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));
38        n = sendto(sock, buffer, strlen(buffer), 0,
39            (struct sockaddr *)&addr, sizeof(addr));
40        if (n < 0) {
41            perror("sendto_error");
42            exit(-1);
43        }
44    }
45    /* normal exit */
46    exit(0);
47 }

```

Figura 17.3: Sezione principale del server per il servizio *daytime* su UDP.

che è stato *legato* ad essi con `bind`, qualora questo sia il caso scriverà il contenuto all'interno del socket, così che il programma possa leggerlo, altrimenti risponderà alla macchina che ha inviato il pacchetto con un messaggio ICMP di tipo *port unreachable*.

Una volta completata la fase di inizializzazione inizia il corpo principale (24-44) del server, mantenuto all'interno di un ciclo infinito in cui si trattano le richieste. Il ciclo inizia (26) con una chiamata a `recvfrom`, che si bloccherà in attesa di pacchetti inviati dai client. Lo scopo della funzione è quello di ritornare tutte le volte che un pacchetto viene inviato al server, in modo da poter ricavare da esso l'indirizzo del client a cui inviare la risposta in `addr`. Per questo motivo in questo caso (al contrario di quanto fatto in fig. 17.2) si è avuto cura di passare gli argomenti `addr` e `len` alla funzione. Dopo aver controllato (27-30) la presenza di eventuali errori (uscendo con un messaggio di errore qualora ve ne siano) si verifica (31) se è stata attivata l'opzione `-v` (che imposta la variabile `verbose`) stampando nel caso (32-35) l'indirizzo da cui si è appena ricevuto una richiesta (questa sezione è identica a quella del server TCP illustrato in fig. 15.10).

Una volta ricevuta la richiesta resta solo da ottenere il tempo corrente (36) e costruire (37) la stringa di risposta, che poi verrà inviata (38) al client usando `sendto`, avendo al solito cura di controllare (40-42) lo stato di uscita della funzione e trattando opportunamente la condizione di errore.

Si noti come per le peculiarità del protocollo si sia utilizzato un server iterativo, che processa le richieste una alla volta via via che gli arrivano. Questa è una caratteristica comune dei server UDP, conseguenza diretta del fatto che non esiste il concetto di connessione, per cui non c'è la necessità di trattare separatamente le singole connessioni. Questo significa anche che è il kernel a gestire la possibilità di richieste multiple in contemporanea; quello che succede è semplicemente che il kernel accumula in un buffer in ingresso i pacchetti UDP che arrivano e li restituisce al processo uno alla volta per ciascuna chiamata di `recvfrom`; nel nostro caso sarà poi compito del server distribuire le risposte sulla base dell'indirizzo da cui provengono le richieste.

17.1.5 Le problematiche dei socket UDP

L'esempio del servizio *daytime* illustrato nelle precedenti sezioni è in realtà piuttosto particolare, e non evidenzia quali possono essere i problemi collegati alla mancanza di affidabilità e all'assenza del concetto di connessione che sono tipiche dei socket UDP. In tal caso infatti il protocollo è estremamente semplice, dato che la comunicazione consiste sempre in una richiesta seguita da una risposta, per uno scambio di dati effettuabile con un singolo pacchetto, per cui tutti gli eventuali problemi sarebbero assai più complessi da rilevare.

Anche qui però possiamo notare che se il pacchetto di richiesta del client, o la risposta del server si perdono, il client resterà permanentemente bloccato nella chiamata a `recvfrom`. Per evidenziare meglio quali problemi si possono avere proviamo allora con un servizio leggermente più complesso come *echo*.

In fig. 17.4 è riportato un estratto del corpo principale del nostro client elementare per il servizio *echo* (al solito il codice completo è con i sorgenti allegati). Le uniche differenze con l'analogo client visto in fig. 15.11 sono che al solito si crea (14) un socket di tipo `SOCK_DGRAM`, e che non è presente nessuna chiamata a `connect`. Per il resto il funzionamento del programma è identico, e tutto il lavoro viene effettuato attraverso la chiamata (28) alla funzione `ClientEcho` che stavolta però prende un argomento in più, che è l'indirizzo del socket.

Ovviamente in questo caso il funzionamento della funzione, il cui codice è riportato in fig. 17.5, è completamente diverso rispetto alla analoga del server TCP, e dato che non esiste

```

1 void ClientEcho(FILE * filein, int socket, struct sockaddr_in *serv_add);
2 void SigTERM_hand(int sig);
3
4 /* Program begin */
5 int main(int argc, char *argv[])
6 {
7     /*
8      * Variables definition
9      */
10     int sock, i;
11     struct sockaddr_in serv_add;
12     ...
13     /* create socket */
14     if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
15         perror("Socket_creation_error");
16         return 1;
17     }
18     /* initialize address */
19     memset((void *) &serv_add, 0, sizeof(serv_add)); /* clear server address */
20     serv_add.sin_family = AF_INET;                  /* address type is INET */
21     serv_add.sin_port = htons(7);                   /* echo port is 7 */
22     /* build address using inet_pton */
23     if ( (inet_pton(AF_INET, argv[optind], &serv_add.sin_addr)) <= 0) {
24         perror("Address_creation_error");
25         return 1;
26     }
27     /* do read/write operations */
28     ClientEcho(stdin, sock, &serv_add);
29     /* normal exit */
30     return 0;
31 }

```

Figura 17.4: Sezione principale della prima versione client per il servizio *echo* su UDP.

```

1 void ClientEcho(FILE * filein, int socket, struct sockaddr_in * serv_addr)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread, nwrite;
5     /* initialize file descriptor set */
6     while (1) {
7         if (fgets(sendbuff, MAXLINE, filein) == NULL) {
8             return; /* if no input just return */
9         } else { /* else we have to write to socket */
10             nwrite = sendto(socket, sendbuff, strlen(sendbuff), 0,
11                             (struct sockaddr *) serv_addr, sizeof(*serv_addr));
12             if (nwrite < 0) { /* on error stop */
13                 printf("Errore_in_scrittura:_%s", strerror(errno));
14                 return;
15             }
16         }
17         nread = recvfrom(socket, recvbuff, strlen(sendbuff), 0, NULL, NULL);
18         if (nread < 0) { /* error condition, stop client */
19             printf("Errore_in_lettura:_%s\n", strerror(errno));
20             return;
21         }
22         recvbuff[nread] = 0; /* else read is ok, write on stdout */
23         if (fputs(recvbuff, stdout) == EOF) {
24             perror("Errore_in_scrittura_su_terminale");
25             return;
26         }
27     }
28 }

```

Figura 17.5: Codice della funzione ClientEcho usata dal client per il servizio *echo* su UDP.

una connessione questa necessita anche di un terzo argomento, che è l'indirizzo del server cui inviare i pacchetti.

Data l'assenza di una connessione come nel caso di TCP il meccanismo è molto più semplice da gestire. Al solito si esegue un ciclo infinito (6-30) che parte dalla lettura (7) sul buffer di invio `sendbuff` di una stringa dallo standard input, se la stringa è vuota (7-9), indicando che l'input è terminato, si ritorna immediatamente causando anche la susseguente terminazione del programma.

Altrimenti si procede (10-11) all'invio della stringa al destinatario invocando `sendto`, utilizzando, oltre alla stringa appena letta, gli argomenti passati nella chiamata a `ClientEcho`, ed in particolare l'indirizzo del server che si è posto in `serv_addr`; qualora (12) si riscontrasse un errore si provvederà al solito (13-14) ad uscire con un messaggio di errore.

Il passo immediatamente seguente (17) l'invio è quello di leggere l'eventuale risposta del server con `recvfrom`; si noti come in questo caso si sia scelto di ignorare l'indirizzo dell'eventuale pacchetto di risposta, controllando (18-21) soltanto la presenza di un errore (nel qual caso al solito si ritorna dopo la stampa di un adeguato messaggio). Si noti anche come, rispetto all'analogia funzione `ClientEcho` utilizzata nel client TCP illustrato in sez. 15.4.2 non si sia controllato il caso di un messaggio nullo, dato che, nel caso di socket UDP, questo non significa la terminazione della comunicazione.

L'ultimo passo (17) è quello di terminare opportunamente la stringa di risposta nel relativo buffer per poi provvedere alla sua stampa sullo standard output, eseguendo il solito controllo (ed eventuale uscita con adeguato messaggio informativo) in caso di errore.

In genere fintanto che si esegue il nostro client in locale non sorgerà nessun problema, se però si proverà ad eseguirlo attraverso un collegamento remoto (nel caso dell'esempio seguente su una VPN, attraverso una ADSL abbastanza congestionata) e in modalità non interattiva, la probabilità di perdere qualche pacchetto aumenta, ed infatti, eseguendo il comando come:

```
[piccardi@gont sources]$ cat UDP_echo.c | ./echo 192.168.1.120
/* UDP_echo.c
 *
 * Copyright (C) 2004 Simone Piccardi
...
...
/*
 * Include needed headers
```

si otterrà che, dopo aver correttamente stampato alcune righe, il programma si blocca completamente senza stampare più niente. Se al contempo si fosse tenuto sotto controllo il traffico UDP diretto o proveniente dal servizio *echo* con `tcpdump` si sarebbe ottenuto:

```
[root@gont gapil]# tcpdump \(\ dst port 7 or src port 7 \)
...
...
18:48:16.390255 gont.earthsea.ea.32788 > 192.168.1.120.echo: udp 4 (DF)
18:48:17.177613 192.168.1.120.echo > gont.earthsea.ea.32788: udp 4 (DF)
18:48:17.177790 gont.earthsea.ea.32788 > 192.168.1.120.echo: udp 26 (DF)
18:48:17.964917 192.168.1.120.echo > gont.earthsea.ea.32788: udp 26 (DF)
18:48:17.965408 gont.earthsea.ea.32788 > 192.168.1.120.echo: udp 4 (DF)
```

che come si vede il traffico fra client e server si interrompe dopo l'invio di un pacchetto UDP per il quale non si è ricevuto risposta.

Il problema è che in tutti i casi in cui un pacchetto di risposta si perde, o una richiesta non arriva a destinazione, il nostro programma si bloccherà nell'esecuzione di `recvfrom`. Lo stesso avviene anche se il server non è in ascolto, in questo caso però, almeno dal punto di vista dello scambio di pacchetti, il risultato è diverso, se si lancia al solito il programma e si prova a scrivere qualcosa si avrà ugualmente un blocco su `recvfrom` ma se si osserva il traffico con `tcpdump` si vedrà qualcosa del tipo:

```
[root@gont gapil]# tcpdump \(( dst 192.168.0.2 and src 192.168.1.120 ) \
or \(( src 192.168.0.2 and dst 192.168.1.120 ) \
tcpdump: listening on eth0
00:43:27.606944 gont.earthsea.ea.32789 > 192.168.1.120.echo: udp 6 (DF)
00:43:27.990560 192.168.1.120 > gont.earthsea.ea: icmp: 192.168.1.120 udp port
echo unreachable [tos 0xc0]
```

cioè in questo caso si avrà in risposta un pacchetto ICMP di destinazione irraggiungibile che ci segnala che la porta in questione non risponde.

Ci si può chiedere allora perché, benché la situazione di errore sia rilevabile, questa non venga segnalata. Il luogo più naturale in cui riportarla sarebbe la chiamata di `sendto`, in quanto è a causa dell'uso di un indirizzo sbagliato che il pacchetto non può essere inviato; farlo in questo punto però è impossibile, dato che l'interfaccia di programmazione richiede che la funzione ritorni non appena il kernel invia il pacchetto,⁹ e non può bloccarsi in una attesa di una risposta che potrebbe essere molto lunga (si noti infatti che il pacchetto ICMP arriva qualche decimo di secondo più tardi) o non esserci affatto.

Si potrebbe allora pensare di riportare l'errore nella `recvfrom` che è comunque bloccata in attesa di una risposta che nel caso non arriverà mai. La ragione per cui non viene fatto è piuttosto sottile e viene spiegata da Stevens in [?] con il seguente esempio: si consideri un client che invia tre pacchetti a tre diverse macchine, due dei quali vengono regolarmente ricevuti, mentre al terzo, non essendo presente un server sulla relativa macchina, viene risposto con un messaggio ICMP come il precedente. Detto messaggio conterrà anche le informazioni relative ad indirizzo e porta del pacchetto che ha fallito, però tutto quello che il kernel può restituire al programma è un codice di errore in `errno`, con il quale è impossibile di distinguere per quale dei pacchetti inviati si è avuto l'errore; per questo è stata fatta la scelta di non riportare un errore su un socket UDP, a meno che, come vedremo in sez. 17.1.6, questo non sia connesso.

17.1.6 L'uso della funzione `connect` con i socket UDP

Come illustrato in sez. 17.1.1 essendo i socket UDP privi di connessione non è necessario per i client usare `connect` prima di iniziare una comunicazione con un server. Ciò non di meno abbiamo accennato come questa possa essere utilizzata per gestire la presenza di errori asincroni.

Quando si chiama `connect` su di un socket UDP tutto quello che succede è che l'indirizzo passato alla funzione viene registrato come indirizzo di destinazione del socket. A differenza di quanto avviene con TCP non viene scambiato nessun pacchetto, tutto quello che succede è che da quel momento in qualunque cosa si scriva sul socket sarà inviata a quell'indirizzo; non sarà più necessario usare l'argomento `to` di `sendto` per specificare la destinazione dei

⁹questo è il classico caso di *errore asincrono*, una situazione cioè in cui la condizione di errore viene rilevata in maniera asincrona rispetto all'operazione che l'ha causata, una eventualità piuttosto comune quando si ha a che fare con la rete, tutti i pacchetti ICMP che segnalano errori rientrano in questa tipologia.

pacchetti, che potranno essere inviati e ricevuti usando le normali funzioni `read` e `write` che a questo punto non falliranno più con l'errore di `EDESTADDRREQ`.¹⁰

Una volta che il socket è connesso cambia però anche il comportamento in ricezione; prima infatti il kernel avrebbe restituito al socket qualunque pacchetto ricevuto con un indirizzo di destinazione corrispondente a quello del socket, senza nessun controllo sulla sorgente; una volta che il socket viene connesso saranno riportati su di esso solo i pacchetti con un indirizzo sorgente corrispondente a quello a cui ci si è connessi.

Infine quando si usa un socket connesso, venendo meno l'ambiguità segnalata alla fine di sez. 17.1.5, tutti gli eventuali errori asincroni vengono riportati alle funzioni che operano su di esso; pertanto potremo riscrivere il nostro client per il servizio *echo* con le modifiche illustrate in fig. 17.6.

```

1 void ClientEcho(FILE * filein, int socket);
2 /* Program begin */
3 int main(int argc, char *argv[])
4 {
5     /*
6      * Variables definition
7      */
8     int sock, i;
9     struct sockaddr_in dst_addr;
10    ...
11    /* create socket */
12    if ( (sock = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
13        perror("Socket_creation_error");
14        return 1;
15    }
16    /* initialize address */
17    memset((void *) &dst_addr, 0, sizeof(dst_addr)); /* clear address */
18    dst_addr.sin_family = AF_INET; /* address type is INET */
19    dst_addr.sin_port = htons(7); /* echo port is 7 */
20    /* build address using inet_pton */
21    if ( (inet_pton(AF_INET, argv[optind], &dst_addr.sin_addr)) <= 0) {
22        perror("Address_creation_error");
23        return 1;
24    }
25    connect(sock, (struct sockaddr *) &dst_addr, sizeof(dst_addr));
26    /* do read/write operations */
27    ClientEcho(stdin, sock);
28    /* normal exit */
29    return 0;
30 }

```

Figura 17.6: Seconda versione del client del servizio *echo* che utilizza socket UDP connessi.

Ed in questo caso rispetto alla precedente versione, il solo cambiamento è l'utilizzo (17) della funzione `connect` prima della chiamata alla funzione di gestione del protocollo, che a sua volta è stata modificata eliminando l'indirizzo passato come argomento e sostituendo le chiamate a `sendto` e `recvfrom` con chiamate a `read` e `write` come illustrato dal nuovo codice riportato in fig. 17.7.

¹⁰in realtà si può anche continuare ad usare la funzione `sendto`, ma in tal caso l'argomento `to` deve essere inizializzato a `NULL`, e `tolen` deve essere inizializzato a zero, pena un errore di `EISCONN`.

```
1 void ClientEcho(FILE * filein, int socket)
2 {
3     char sendbuff[MAXLINE+1], recvbuff[MAXLINE+1];
4     int nread, nwrite;
5     /* initialize file descriptor set */
6     while (1) {
7         if (fgets(sendbuff, MAXLINE, filein) == NULL) {
8             return; /* if no input just return */
9         } else { /* else we have to write to socket */
10             nwrite = write(socket, sendbuff, strlen(sendbuff));
11             if (nwrite < 0) { /* on error stop */
12                 printf("Errore_in_scrittura:_%s", strerror(errno));
13                 return;
14             }
15         }
16         nread = read(socket, recvbuff, strlen(sendbuff));
17         if (nread < 0) { /* error condition, stop client */
18             printf("Errore_in_lettura:_%s\n", strerror(errno));
19             return;
20         }
21         recvbuff[nread] = 0; /* else read is ok, write on stdout */
22         if (fputs(recvbuff, stdout) == EOF) {
23             perror("Errore_in_scrittura_su_terminale");
24             return;
25         }
26     }
27 }
```

Figura 17.7: Seconda versione della funzione ClientEcho.

Utilizzando questa nuova versione del client si può verificare che quando ci si rivolge verso un indirizzo inesistente o su cui non è in ascolto un server si è in grado rilevare l'errore, se infatti eseguiamo il nuovo programma otterremo un qualcosa del tipo:

```
[piccardi@gont sources]$ ./echo 192.168.1.1  
prova  
Errore in lettura: Connection refused
```

Ma si noti che a differenza di quanto avveniva con il client TCP qui l'errore viene rilevato soltanto dopo che si è tentato di inviare qualcosa, ed in corrispondenza al tentativo di lettura della risposta. Questo avviene perché con UDP non esiste una connessione, e fintanto che non si invia un pacchetto non c'è traffico sulla rete. In questo caso l'errore sarà rilevato alla ricezione del pacchetto ICMP *destination unreachable* emesso dalla macchina cui ci si è rivolti, e questa volta, essendo il socket UDP connesso, il kernel potrà riportare detto errore in *user space* in maniera non ambigua, ed esso apparirà alla successiva lettura sul socket.

Si tenga presente infine che l'uso dei socket connessi non risolve l'altro problema del client, e cioè il fatto che in caso di perdita di un pacchetto questo resterà bloccato permanentemente in attesa di una risposta. Per risolvere questo problema l'unico modo sarebbe quello di impostare un *timeout* o riscrivere il client in modo da usare l'I/O non bloccante.

17.2 I socket *Unix domain*

Benché i socket Unix domain, come meccanismo di comunicazione fra processi che girano sulla stessa macchina, non siano strettamente attinenti alla rete, li tratteremo comunque in questa sezione. Nonostante le loro peculiarità infatti, l'interfaccia di programmazione che serve ad utilizzarli resta sempre quella dei socket.

17.2.1 Il passaggio di file descriptor

17.3 Altri socket

Tratteremo in questa sezione gli altri tipi particolari di socket supportati da Linux, come quelli relativi a particolari protocolli di trasmissione, i socket *netlink* che definiscono una interfaccia di comunicazione con il kernel, ed i *packet socket* che consentono di inviare pacchetti direttamente a livello delle interfacce di rete.

17.3.1 I socket *raw*

Tratteremo in questa sezione i cosiddetti *raw socket*, con i quali si possono *forgiare* direttamente i pacchetti a tutti i livelli dello stack dei protocolli.

17.3.2 I socket *netlink*

17.3.3 I *packet socket*

Capitolo 18

Socket avanzati

Esamineremo in questo capitolo le funzionalità più evolute della gestione dei socket, le funzioni avanzate, la gestione dei dati urgenti e *out-of-band* e dei messaggi ancillari, come l'uso come l'uso del I/O multiplexing (vedi sez. 10.2) con i socket.

18.1 Le funzioni di I/O avanzate

Tratteremo in questa sezione le funzioni di I/O più avanzate che permettono di controllare le funzionalità specifiche della comunicazione dei dati che sono disponibili con i vari tipi di socket.

18.1.1 La funzioni `send` e `recv`

Da fare

18.1.2 La funzioni `sendmsg` e `recvmsg`

Finora abbiamo trattato delle funzioni che permettono di inviare dati sul socket in forma semplificata. Se infatti si devono semplicemente ...

18.1.3 I messaggi ancillari

Quanto è stata attivata l'opzione `IP_RECVERR` il kernel attiva per il socket una speciale coda su cui vengono inviati tutti gli errori riscontrati. Questi possono essere riletti usando il flag `MSG_ERRQUEUE`, nel qual caso sarà passato come messaggio ancillare una struttura di tipo `sock_extended_err` illustrata in fig. 18.1.

18.1.4 I *dati urgenti* o *out-of-band*

Una caratteristica particolare dei socket TCP è quella che consente di inviare all'altro capo della comunicazione una sorta di messaggio privilegiato, che si richiede che sia trattato il prima possibile. Si fa riferimento a questa funzionalità come all'invio dei cosiddetti *dati urgenti* (o *urgent data*); talvolta essi chiamati anche dati *out-of-band* poiché, come vedremo più avanti, possono essere letti anche al di fuori del flusso di dati normale.

```

struct sock_extended_err {
    u_int32_t    ee_errno;    /* error number */
    u_int8_t     ee_origin;   /* where the error originated */
    u_int8_t     ee_type;     /* type */
    u_int8_t     ee_code;     /* code */
    u_int8_t     ee_pad;
    u_int32_t     ee_info;     /* additional information */
    u_int32_t     ee_data;     /* other data */
    /* More data may follow */
};

```

Figura 18.1: La struttura `sock_extended_err` usata dall'opzione `IP_RECVERR` per ottenere le informazioni relative agli errori su un socket.

Come già accennato in sez. 10.2 la presenza di dati urgenti viene rilevata in maniera specifica sia di `select` (con il *file descriptor set exceptfds*) che da `poll` (con la condizione `POLLRDBAND`).

Le modalità di lettura dei dati urgenti sono due, la prima e più comune prevede l'uso di `recvmsg` con

La seconda modalità di lettura prevede invece l'uso dell'opzione dei socket `SO_OOBINLINE` (vedi sez. 16.2.2) che consente di ricevere i dati urgenti direttamente nel flusso dei dati del socket; in tal caso però si pone il problema di come distinguere i dati normali da quelli urgenti. Come già accennato in sez. 16.3.3 a questo scopo si può usare `ioctl` con l'operazione `SIOCATMARK`, che consente di sapere se si è arrivati o meno all'*urgent mark*.

La procedura allora prevede che, una volta che si sia rilevata la presenza di dati urgenti, si ripeta la lettura ordinaria dal socket fintanto che `SIOCATMARK` non restituisce un valore diverso da zero; la successiva lettura restituirà i dati urgenti.

18.2 L'uso dell'I/O non bloccante

Tratteremo in questa sezione le modalità avanzate che permettono di utilizzare i socket con una comunicazione non bloccante, in modo da

18.2.1 La gestione delle opzioni IP

Abbiamo visto in sez. 16.2.4 come di possa usare `setsockopt` con l'opzione `IP_OPTIONS` per impostare le opzioni IP associate per i pacchetti associati ad un socket. Vedremo qui il significato di tali opzioni e le modalità con cui esse possono essere utilizzate ed impostate.

Parte III

Appendici

Appendice A

I codici di errore

Si riportano in questa appendice tutti i codici di errore. Essi sono accessibili attraverso l'inclusione del file di header `errno.h`, che definisce anche la variabile globale `errno`. Per ogni errore definito riporteremo la stringa stampata da `perror` ed una breve spiegazione. Si tenga presente che spiegazioni più particolareggiate del significato dell'errore, qualora necessarie per casi specifici, possono essere trovate nella descrizione del prototipo della funzione per cui detto errore si è verificato.

I codici di errore sono riportati come costanti di tipo `int`, i valori delle costanti sono definiti da macro di preprocessore nel file citato, e possono variare da architettura a architettura; è pertanto necessario riferirsi ad essi tramite i nomi simbolici. Le funzioni `perror` e `strerror` (vedi sez. 6.5.2) possono essere usate per ottenere dei messaggi di errore più espliciti.

A.1 Gli errori dei file

In questa sezione sono raccolti i codici restituiti dalle *system call* attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei file.

EACCES *Permission denied.*

Permesso negato; l'accesso al file o alla directory non è consentito: i permessi del file o della directory o quelli necessari ad attraversare un *pathname* non consentono l'operazione richiesta.

EBADF *Bad file descriptor.*

File descriptor non valido: si è usato un file descriptor inesistente, o aperto in sola lettura per scrivere, o viceversa, o si è cercato di eseguire un'operazione non consentita per quel tipo di file descriptor.

EBUSY *Resource busy.*

Una risorsa di sistema che non può essere condivisa è occupata. Ad esempio si è tentato di cancellare la directory su cui si è montato un filesystem.

EDQUOT *Quota exceeded.*

Si è ecceduta la quota di disco dell'utente, usato sia per lo spazio disco che per il numero di *inode*.

EEXIST *File exists.*

Si è specificato un file esistente in un contesto in cui ha senso solo specificare un nuovo file.

EFBIG *File too big.*

Si è ecceduto il limite imposto dal sistema sulla dimensione massima che un file può avere.

EFTYPE *Inappropriate file type or format.*

Il file è di tipo sbagliato rispetto all'operazione richiesta o un file di dati ha un formato sbagliato. Alcuni sistemi restituiscono questo errore quando si cerca di impostare lo *sticky bit* su un file che non è una directory.

EIO *Input/output error.*

Errore di input/output: usato per riportare errori hardware in lettura/scrittura su un dispositivo.

EISDIR *Is a directory.*

Il file specificato è una directory; non può essere aperto in scrittura, né si possono creare o rimuovere link diretti ad essa.

ELOOP *Too many symbolic links encountered.*

Ci sono troppi link simbolici nella risoluzione di un *pathname*.

EMFILE *Too many open files.*

Il processo corrente ha troppi file aperti e non può aprirne altri. Anche i descrittori duplicati ed i socket vengono tenuti in conto.¹

EMLINK *Too many links.*

Ci sono già troppi link al file, il numero massimo è specificato dalla variabile `LINK_MAX`, vedi sez. 6.1.1.

ENAMETOOLONG *File name too long.*

Si è indicato un *pathname* troppo lungo per un file o una directory.

ENFILE *File table overflow.*

Il sistema ha troppi file aperti in contemporanea. Si tenga presente che anche i socket contano come file. Questa è una condizione temporanea, ed è molto difficile che si verifichi nei sistemi moderni.

ENODEV *No such device.*

Si è indicato un tipo di device sbagliato ad una funzione che ne richiede uno specifico.

ENOENT *No such file or directory.*

Il file indicato dal *pathname* non esiste: o una delle componenti non esiste o il *pathname* contiene un link simbolico spezzato. Errore tipico di un riferimento ad un file che si suppone erroneamente essere esistente.

ENOEXEC *Invalid executable file format.*

Il file non ha un formato eseguibile, è un errore riscontrato dalle funzioni `exec`.

ENOLCK *No locks available.*

È usato dalle utilità per la gestione del file locking; non viene generato da un sistema GNU, ma può risultare da un'operazione su un server NFS di un altro sistema.

¹il numero massimo di file aperti è controllabile dal sistema; in Linux si può impostare usando il comando `ulimit`, esso è in genere indicato dalla costante `OPEN_MAX`, vedi sez. 6.1.1.

ENOSPC *No space left on device.*

La directory in cui si vuole creare il link non ha spazio per ulteriori voci, o si è cercato di scrivere o di creare un nuovo file su un dispositivo che è già pieno.

ENOTBLK *Block device required.*

Si è specificato un file che non è un *block device* in un contesto in cui era necessario specificare un *block device* (ad esempio si è tentato di montare un file ordinario).

ENOTDIR *Not a directory.*

Si è specificato un file che non è una directory in una operazione che richiede una directory.

ENOTEMPTY *Directory not empty.*

La directory non è vuota quando l'operazione richiede che lo sia. È l'errore tipico che si ha quando si cerca di cancellare una directory contenente dei file.

ENOTTY *Not a terminal.*

Si è tentata una operazione di controllo relativa ad un terminale su un file che non lo è.

ENXIO *No such device or address.*

Dispositivo inesistente: il sistema ha tentato di usare un dispositivo attraverso il file specificato, ma non lo ha trovato. Può significare che il file di dispositivo non è corretto, che il modulo relativo non è stato caricato nel kernel, o che il dispositivo è fisicamente assente o non funzionante.

EPERM *Operation not permitted.*

L'operazione non è permessa: solo il proprietario del file o un processo con sufficienti privilegi può eseguire l'operazione.

EPIPE *Broken pipe.*

Non c'è un processo che stia leggendo l'altro capo della *pipe*. Ogni funzione che restituisce questo errore genera anche un segnale SIGPIPE, la cui azione predefinita è terminare il programma; pertanto non si potrà vedere questo errore fintanto che SIGPIPE non viene gestito o bloccato.

EREMOTE *Object is remote.*

Si è fatto un tentativo di montare via NFS un filesystem remoto con un nome che già specifica un filesystem montato via NFS.

EROFS *Read-only file system.*

Si è cercato di eseguire una operazione di scrittura su un file o una directory che risiede su un filesystem montato in sola lettura.

ESPIPE *Invalid seek operation.*

Si è cercato di eseguire una *lseek* su un file che non supporta questa operazione (ad esempio su una *pipe*, da cui il nome).

ESTALE *Stale file handle.*

Indica un problema interno ad un filesystem remoto (in genere con NFS) causato da cambiamenti del filesystem del sistema remoto. Per recuperare questa condizione in genere è necessario smontare e rimontare il filesystem.

ETXTBSY *Text file busy.*

Si è cercato di eseguire un file che è aperto in scrittura, o di scrivere su un file che è in esecuzione.

EUSERS *Too many users.*

Troppi utenti, il sistema delle quote rileva troppi utenti nel sistema.

EXDEV *Cross-device link.*

Si è tentato di creare un link diretto che attraversa due filesystem differenti.

A.2 Gli errori dei processi

In questa sezione sono raccolti i codici restituiti dalle *system call* attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei processi.

E2BIG *Argument list too long.*

La lista degli argomenti passati è troppo lunga: è una condizione prevista da POSIX quando la lista degli argomenti passata ad una delle funzioni `exec` occupa troppa memoria.

ECHILD *There are no child processes.*

Non esistono processi figli di cui attendere la terminazione. Viene rilevato dalle funzioni `wait` e `waitpid` (vedi sez. 3.1.5).

EPROCLIM *Too many processes.*

Il limite dell'utente per nuovi processi (vedi sez. 6.3.2) sarà ecceduto alla prossima `fork`; è un codice di errore di BSD, che non viene utilizzato al momento su Linux.

ESRCH *No process matches the specified process ID.*

Non esiste un processo o un *process group* corrispondenti al valore dell'identificativo specificato.

A.3 Gli errori di rete

In questa sezione sono raccolti i codici restituiti dalle *system call* attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei socket e delle connessioni di rete.

EADDRINUSE *Address already in use.*

L'indirizzo del socket richiesto è già utilizzato (ad esempio si è eseguita `bind` su una porta già in uso).

EADDRNOTAVAIL *Cannot assign requested address.*

L'indirizzo richiesto non è disponibile (ad esempio si è cercato di dare al socket un nome che non corrisponde al nome della stazione locale), o l'interfaccia richiesta non esiste.

EAFNOSUPPORT *Address family not supported by protocol.*

Famiglia di indirizzi non supportata. La famiglia di indirizzi richiesta non è supportata, o è inconsistente con il protocollo usato dal socket.

ECONNABORTED *Software caused connection abort.*

Una connessione è stata abortita localmente.

ECONNREFUSED *Connection refused.*

Un host remoto ha rifiutato la connessione (in genere dipende dal fatto che non c'è un server per soddisfare il servizio richiesto).

ECONNRESET *Connection reset by peer.*

Una connessione è stata chiusa per ragioni fuori dal controllo dell'host locale, come il riavvio di una macchina remota o un qualche errore non recuperabile sul protocollo.

EDESTADDRREQ *Destination address required.*

Non c'è un indirizzo di destinazione predefinito per il socket. Si ottiene questo errore mandando dato su un socket senza connessione senza averne prima specificato una destinazione.

EHOSTDOWN *Host is down.*

L'host remoto non risponde.

EHOSTUNREACH *No route to host.*

L'host remoto non è raggiungibile. Si ottiene quando non esiste una rotta di rete per arrivare all'host remoto con il protocollo di rete utilizzato.

EISCONN *Transport endpoint is already connected.*

Si è tentato di connettere un socket che è già connesso.

EMSGSIZE *Message too long.*

Le dimensioni di un messaggio inviato su un socket sono eccedono la massima lunghezza supportata.

ENETDOWN *Network is down.*

L'operazione sul socket è fallita perché la rete è sconnessa.

ENETRESET *Network dropped connection because of reset.*

Una connessione è stata cancellata perché l'host remoto è caduto.

ENETUNREACH *Network is unreachable.*

L'operazione è fallita perché l'indirizzo richiesto è irraggiungibile (ad esempio la sottorete della stazione remota è irraggiungibile).

ENOBUFS *No buffer space available.*

Tutti i buffer per le operazioni di I/O del kernel sono occupati. In generale questo errore è sinonimo di ENOMEM, ma attiene alle funzioni di input/output. In caso di operazioni sulla rete si può ottenere questo errore invece dell'altro.

ENOPROTOPT *Protocol not available.*

Protocollo non disponibile. Si è richiesta un'opzione per il socket non disponibile con il protocollo usato.

ENOTCONN *Transport endpoint is not connected.*

Il socket non è connesso a niente. Si ottiene questo errore quando si cerca di trasmettere dati su un socket senza avere specificato in precedenza la loro destinazione. Nel caso di socket senza connessione (ad esempio socket UDP) l'errore che si ottiene è EDESTADDRREQ.

ENOTSOCK *Socket operation on non-socket.*

Si è tentata un'operazione su un file descriptor che non è un socket quando invece era richiesto un socket.

EOPNOTSUPP *Operation not supported on transport endpoint.*

L'operazione richiesta non è supportata. Alcune funzioni non hanno senso per tutti i tipi di socket, ed altre non sono implementate per tutti i protocolli di trasmissione. Questo errore quando un socket non supporta una particolare operazione, e

costituisce una indicazione generica che il server non sa cosa fare per la chiamata effettuata.

EPFNOSUPPORT *Protocol family not supported.*

Famiglia di protocolli non supportata. La famiglia di protocolli richiesta non è supportata.

EPROTONOSUPPORT *Protocol not supported.*

Protocollo non supportato. Il tipo di socket non supporta il protocollo richiesto (un probabile errore nella specificazione del protocollo).

EPROTOTYPE *Protocol wrong type for socket.*

Protocollo sbagliato per il socket. Il socket usato non supporta il protocollo di comunicazione richiesto.

ESHUTDOWN *Cannot send after transport endpoint shutdown.*

Il socket su cui si cerca di inviare dei dati ha avuto uno shutdown.

ESOCKTNOSUPPORT *Socket type not supported.*

Socket non supportato. Il tipo di socket scelto non è supportato.

ETIMEDOUT *Connection timed out.*

Un'operazione sul socket non ha avuto risposta entro il periodo di timeout.

ETOOMANYREFS *Too many references: cannot splice.*

La *glibc* dice ???

A.4 Errori generici

In questa sezione sono raccolti i codici restituiti dalle *system call* attinenti ad errori generici, si trovano qui tutti i codici di errore non specificati nelle sezioni precedenti.

EAGAIN *Resource temporarily unavailable.*

La funzione è fallita ma potrebbe funzionare se la chiamata fosse ripetuta. Questo errore accade in due tipologie di situazioni:

- Si è effettuata un'operazione che si sarebbe bloccata su un oggetto che è stato posto in modalità non bloccante. Nei vecchi sistemi questo era un codice diverso, **EWOULDBLOCK**. In genere questo ha a che fare con file o socket, per i quali si può usare la funzione **select** per vedere quando l'operazione richiesta (lettura, scrittura o connessione) diventa possibile.
- Indica la carenza di una risorsa di sistema che non è al momento disponibile (ad esempio **fork** può fallire con questo errore se si è esaurito il numero di processi contemporanei disponibili). La ripetizione della chiamata in un periodo successivo, in cui la carenza della risorsa richiesta può essersi attenuata, può avere successo. Questo tipo di carenza è spesso indice di qualcosa che non va nel sistema, è pertanto opportuno segnalare esplicitamente questo tipo di errori.

EALREADY *Operation already in progress.*

L'operazione è già in corso. Si è tentata un'operazione già in corso su un oggetto posto in modalità non-bloccante.

EDEADLK *Deadlock avoided.*

L'allocazione di una risorsa avrebbe causato un *deadlock*. Non sempre il sistema è in grado di riconoscere queste situazioni, nel qual caso si avrebbe il blocco.

EFAULT *Bad address.*

Una stringa passata come argomento è fuori dello spazio di indirizzi del processo, in genere questa situazione provoca direttamente l'emissione di un segnale di *segment violation* (SIGSEGV).

EDOM *Domain error.*

È usato dalle funzioni matematiche quando il valore di un argomento è al di fuori dell'intervallo in cui esse sono definite.

EILSEQ *Illegal byte sequence.*

Nella decodifica di un carattere esteso si è avuta una sequenza errata o incompleta o si è specificato un valore non valido.

EINPROGRESS *Operation now in progress.*

Operazione in corso. Un'operazione che non può essere completata immediatamente è stata avviata su un oggetto posto in modalità non-bloccante. Questo errore viene riportato per operazioni che si dovrebbero sempre bloccare (come per una *connect*) e che pertanto non possono riportare *EAGAIN*, l'errore indica che l'operazione è stata avviata correttamente e occorrerà del tempo perché si possa completare. La ripetizione della chiamata darebbe luogo ad un errore *EALREADY*.

EINTR *Interrupted function call.*

Una funzione di libreria è stata interrotta. In genere questo avviene causa di un segnale asincrono al processo che impedisce la conclusione della chiamata, la funzione ritorna con questo errore una volta che si sia correttamente eseguito il gestore del segnale. In questo caso è necessario ripetere la chiamata alla funzione.

EINVAL *Invalid argument.*

Errore utilizzato per segnalare vari tipi di problemi dovuti all'aver passato un argomento sbagliato ad una funzione di libreria.

ENOMEM *No memory available.*

Il kernel non è in grado di allocare ulteriore memoria per completare l'operazione richiesta.

ENOSYS *Function not implemented.*

Indica che la funzione non è supportata o nelle librerie del C o nel kernel. Può dipendere sia dalla mancanza di una implementazione, che dal fatto che non si è abilitato l'opportuno supporto nel kernel; nel caso di Linux questo può voler dire anche che un modulo necessario non è stato caricato nel sistema.

ENOTSUP *Not supported.*

Una funzione ritorna questo errore quando gli argomenti sono validi ma l'operazione richiesta non è supportata. Questo significa che la funzione non implementa quel particolare comando o opzione o che, in caso di oggetti specifici (file descriptor o altro) non è in grado di supportare i parametri richiesti.

ERANGE *Range error.*

È usato dalle funzioni matematiche quando il risultato dell'operazione non è rappresentabile nel valore di ritorno a causa di un overflow o di un underflow.

EWouldBlock *Operation would block.*

Indica che l'operazione richiesta si bloccherebbe, ad esempio se si apre un file in modalità non bloccante, una `read` restituirebbe questo errore per indicare che non ci sono dati; in Linux è identico a `EAGAIN`, ma in altri sistemi può essere specificato un valore diverso.

EBadMsg *Not a data message.*

Definito da POSIX come errore che arriva ad una funzione di lettura che opera su uno stream. Non essendo gli stream definiti su Linux il kernel non genera mai questo tipo di messaggio.

EIDRM *Identifier removed.*

Indica che l'oggetto del *SysV IPC* a cui si fa riferimento è stato cancellato.

EMultiHop *Multihop attempted.*

Definito da POSIX come errore dovuto all'accesso a file remoti attraverso più macchine, quando ciò non è consentito. Non viene mai generato su Linux.

ENOATTR *No such attribute.*

È un codice di errore specifico di Linux utilizzato dalle funzioni per la gestione degli attributi estesi dei file (vedi sez. 4.5.1) quando il nome dell'attributo richiesto non viene trovato.

ENODATA *No data available.*

Viene indicato da POSIX come restituito da una `read` eseguita su un file descriptor in modalità non bloccante quando non ci sono dati. In realtà in questo caso su Linux viene utilizzato `EAGAIN`. Lo stesso valore però viene usato come sinonimo di `ENOATTR`.

ENOLINK *Link has been severed.*

È un errore il cui valore è indicato come *riservato* nelle *Single Unix Specification*. Dovrebbe indicare l'impossibilità di accedere ad un file a causa di un errore sul collegamento di rete, ma non ci sono indicazioni precise del suo utilizzo. Per quanto riguarda Linux viene riportato nei sorgenti del kernel in alcune operazioni relative ad operazioni di rete.

ENOMSG *No message of desired type.*

Indica che in una coda di messaggi del *SysV IPC* non è presente nessun messaggio del tipo desiderato.

ENOSR *Out of streams resources.*

Errore relativo agli *STREAMS*, che indica l'assenza di risorse sufficienti a completare l'operazione richiesta. Quella degli *STREAMS*² è interfaccia di programmazione originaria di System V, che non è implementata da Linux, per cui questo errore non viene utilizzato.

ENOSTR *Device not a stream.*

Altro errore relativo agli *STREAMS*, anch'esso non utilizzato da Linux.

EOVERFLOW *Value too large for defined data type.*

Si è chiesta la lettura di un dato dal *SysV IPC* con `IPC_STAT` ma il valore eccede la dimensione usata nel buffer di lettura.

²che non vanno confusi con gli *stream* di sez. 5.3.

EPROTO *Protocol error.*

Indica che c'è stato un errore nel protocollo di rete usato dal socket; viene usato come errore generico dall'interfaccia degli *STREAMS* quando non si è in grado di specificare un altro codice di errore che esprima più accuratamente la situazione.

ETIME *Timer expired.*

Indica che è avvenuto un timeout nell'accesso ad una risorsa (ad esempio un semaforo). Compare nei sorgenti del kernel (in particolare per le funzioni relative al bus USB) come indicazione di una mancata risposta di un dispositivo, con una descrizione alternativa di *Device did not respond*.

Appendice B

Il livello di rete

In questa appendice prenderemo in esame i vari protocolli disponibili a livello di rete.¹ Per ciascuno di essi forniremo una descrizione generica delle principali caratteristiche, del formato di dati usato e quanto possa essere necessario per capirne meglio il funzionamento dal punto di vista della programmazione.

Data la loro prevalenza il capitolo sarà sostanzialmente incentrato sui due protocolli principali esistenti su questo livello: il protocollo IP, sigla che sta per *Internet Protocol*, (ma che più propriamente si dovrebbe chiamare IPv4) ed la nuova versione di questo stesso protocollo, denominata IPv6. Tratteremo comunque anche il protocollo ICMP e la sua versione modificata per IPv6 (cioè ICMPv6).

B.1 Il protocollo IP

L'attuale *Internet Protocol* (IPv4) viene standardizzato nel 1981 dall'RFC 791; esso nasce per disaccoppiare le applicazioni della struttura hardware delle reti di trasmissione, e creare una interfaccia di trasmissione dei dati indipendente dal sottostante substrato di rete, che può essere realizzato con le tecnologie più disparate (Ethernet, Token Ring, FDDI, ecc.).

B.1.1 Introduzione

Il compito principale di IP è quello di trasmettere i pacchetti da un computer all'altro della rete; le caratteristiche essenziali con cui questo viene realizzato in IPv4 sono due:

- *Universal addressing* la comunicazione avviene fra due host identificati univocamente con un indirizzo a 32 bit che può appartenere ad una sola interfaccia di rete.
- *Best effort* viene assicurato il massimo impegno nella trasmissione, ma non c'è nessuna garanzia per i livelli superiori né sulla percentuale di successo né sul tempo di consegna dei pacchetti di dati, né sull'ordine in cui vengono consegnati.

Per effettuare la comunicazione e l'instradamento dei pacchetti fra le varie reti di cui è composta Internet IPv4 organizza gli indirizzi in una gerarchia a due livelli, in cui una parte dei 32 bit dell'indirizzo indica il numero di rete, e un'altra l'host al suo interno. Il numero di

¹per la spiegazione della suddivisione in livelli dei protocolli di rete, si faccia riferimento a quanto illustrato in sez. 13.2.

rete serve ai router per stabilire a quale rete il pacchetto deve essere inviato, il numero di host indica la macchina di destinazione finale all'interno di detta rete.

Per garantire l'unicità dell'indirizzo Internet esiste un'autorità centrale (la IANA, *Internet Assigned Number Authority*) che assegna i numeri di rete alle organizzazioni che ne fanno richiesta; è poi compito di quest'ultime assegnare i numeri dei singoli host all'interno della propria rete.

Per venire incontro alle richieste dei vari enti e organizzazioni che volevano utilizzare questo protocollo di comunicazione, originariamente gli indirizzi di rete erano stati suddivisi all'interno delle cosiddette *classi*, (rappresentate in tab. B.1), in modo da consentire dispiegamenti di reti di varie dimensioni a seconda delle diverse esigenze.

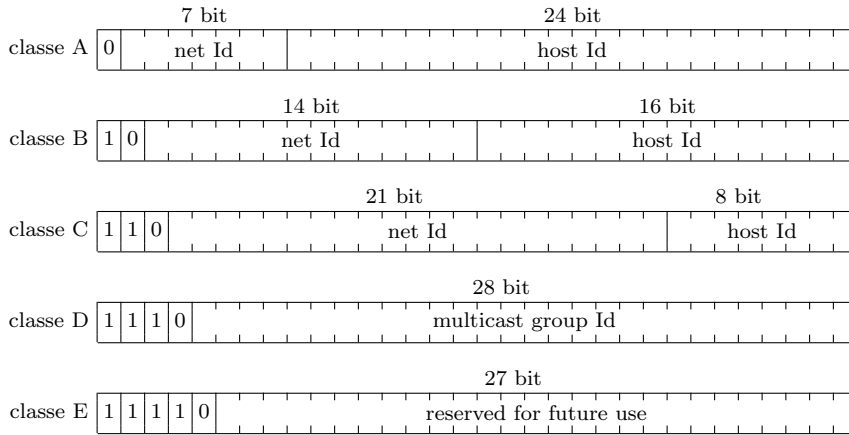


Tabella B.1: Le classi di indirizzi secondo IPv4.

Le classi di indirizzi usate per il dispiegamento delle reti su quella che comunemente viene chiamata *Internet* sono le prime tre; la classe D è destinata al *multicast* mentre la classe E è riservata per usi sperimentali e non viene impiegata.

Come si può notare però la suddivisione riportata in tab. B.1 è largamente inefficiente in quanto se ad un utente necessita anche solo un indirizzo in più dei 256 disponibili con una classe A occorre passare a una classe B, che ne prevede 65536,² con un conseguente spreco di numeri.

Inoltre, in particolare per le reti di classe C, la presenza di tanti indirizzi di rete diversi comporta una crescita enorme delle tabelle di instradamento che ciascun router dovrebbe tenere in memoria per sapere dove inviare il pacchetto, con conseguente crescita dei tempi di elaborazione da parte di questi ultimi ed inefficienza nel trasporto.

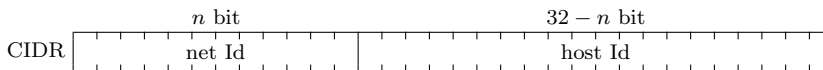


Tabella B.2: Uno esempio di indirizzamento CIDR.

Per questo nel 1992 è stato introdotto un indirizzamento senza classi (il CIDR, *Classless Inter-Domain Routing*) in cui il limite fra i bit destinati a indicare il numero di rete e quello

²in realtà i valori esatti sarebbero 254 e 65534, una rete con a disposizione N bit dell'indirizzo IP, ha disponibili per le singole macchine soltanto $2^N - 2$ numeri, dato che uno deve essere utilizzato come indirizzo di rete e uno per l'indirizzo di *broadcast*.

destinati a indicare l'host finale può essere piazzato in qualunque punto (vedi tab. B.2), permettendo di accorpare più classi A su un'unica rete o suddividere una classe B e diminuendo al contempo il numero di indirizzi di rete da inserire nelle tabelle di instradamento dei router.

B.1.2 L'intestazione di IP

Come illustrato in fig. 13.2 (si ricordi quanto detto in sez. 13.2.2 riguardo al funzionamento generale del TCP/IP), per eseguire il suo compito il protocollo IP inserisce (come praticamente ogni protocollo di rete) una opportuna intestazione in cima ai dati che deve trasmettere, la cui schematizzazione è riportata in fig. B.1.

Figura B.1: L'intestazione o *header* di IPv4.

Ciascuno dei campi illustrati in fig. B.1 ha un suo preciso scopo e significato, che si è riportato brevemente in tab. B.3; si noti come l'intestazione riporti sempre due indirizzi IP, quello *sorgente*, che indica l'IP da cui è partito il pacchetto (cioè l'indirizzo assegnato alla macchina che lo spedisce) e quello *destinazione* che indica l'indirizzo a cui deve essere inviato il pacchetto (cioè l'indirizzo assegnato alla macchina che lo riceverà).

Nome	Bit	Significato
<i>version</i>	4	Numero di <i>versione</i> , nel caso specifico vale sempre 4.
<i>head length</i>	4	Lunghezza dell'intestazione, in multipli di 32 bit.
<i>type of service</i>	8	Il " <i>tipo di servizio</i> ", è suddiviso in: 3 bit di precedenza, che nelle attuali implementazioni del protocollo non vengono comunque utilizzati; un bit riservato che deve essere mantenuto a 0; 4 bit che identificano il tipo di servizio richiesto, uno solo dei quali può essere attivo.
<i>total length</i>	16	La <i>lunghezza totale</i> , indica la dimensione del carico di dati del pacchetto IP in byte.
<i>identification</i>	16	L' <i>identificazione</i> , assegnato alla creazione, è aumentato di uno all'origine della trasmissione di ciascun pacchetto, ma resta lo stesso per i pacchetti frammentati, consentendo così di identificare quelli che derivano dallo stesso pacchetto originario.
<i>flag</i>	3	I <i>flag</i> di controllo nell'ordine: il primo è riservato e sempre nullo, il secondo indica se il pacchetto non può essere frammentato, il terzo se ci sono ulteriori frammenti.
<i>fragmentation offset</i>	13	L' <i>offset di frammento</i> , indica la posizione del frammento rispetto al pacchetto originale.
<i>time to live</i>	16	Il <i>tempo di vita</i> , è decrementato di uno ogni volta che un router ritrasmette il pacchetto, se arriva a zero il pacchetto viene scartato.
<i>protocol</i>	8	Il <i>protocollo</i> , identifica il tipo di pacchetto che segue l'intestazione di IPv4.
<i>header checksum</i>	16	La <i>checksum di intestazione</i> , somma di controllo per l'intestazione.
<i>source IP</i>	32	L' <i>indirizzo di origine</i> .
<i>destination IP</i>	32	L' <i>indirizzo di destinazione</i> .

Tabella B.3: Legenda per il significato dei campi dell'intestazione di IPv4

Il campo TOS definisce il cosiddetto *Type of Service*; questo permette di definire il tipo di traffico contenuto nei pacchetti, e può essere utilizzato dai router per dare diverse priorità in base al valore assunto da questo campo. Abbiamo già visto come il valore di questo campo può essere impostato sul singolo socket con l'opzione IP_TOS (vedi sez. 16.2.4), esso inoltre può essere manipolato sia dal sistema del *netfilter* di Linux con il comando *iptables* che dal sistema del routing avanzato del comando *ip route* per consentire un controllo più dettagliato dell'instradamento dei pacchetti e l'uso di priorità e politiche di distribuzione degli stessi.

Valore		Significato
IPTOS_LOWDELAY	0x10	Minimizza i ritardi per rendere più veloce possibile la trasmissione dei pacchetti (usato per traffico interattivo di controllo come SSH).
IPTOS_THROUGHPUT	0x8	Ottimizza la trasmissione per rendere il più elevato possibile il flusso netto di dati (usato su traffico dati, come quello di FTP).
IPTOS_RELIABILITY	0x4	Ottimizza la trasmissione per ridurre al massimo le perdite di pacchetti (usato su traffico soggetto a rischio di perdita di pacchetti come TFTP o DHCP).
IPTOS_MINCOST	0x2	Indica i dati di riempimento, dove non interessa se si ha una bassa velocità di trasmissione, da utilizzare per i collegamenti con minor costo (usato per i protocolli di streaming).
Normal-Service	0x0	Nessuna richiesta specifica.

Tabella B.4: Le costanti che definiscono alcuni valori standard per il campo TOS da usare come argomento `optval` per l'opzione `IP_TOS`.

I possibili valori del campo TOS, insieme al relativo significato ed alle costanti numeriche ad esso associati, sono riportati in tab. B.4. Per il valore nullo, usato di default per tutti i pacchetti, e relativo al traffico normale, non esiste nessuna costante associata.

Il campo TTL, acronimo di *Time To Live*, viene utilizzato per stabilire una sorta di tempo di vita massimo dei pacchetti sulla rete. In realtà più che di un tempo, il campo serve a limitare il numero massimo di salti (i cosiddetti *hop*) che un pacchetto IP può compiere nel passare da un router ad un altro nel suo attraversamento della rete verso la destinazione.

Il protocollo IP prevede infatti che il valore di questo campo venga decrementato di uno da ciascun router che ritrasmette il pacchetto verso la sua destinazione, e che quando questo diventa nullo il router lo debba scartare, inviando all'indirizzo sorgente un pacchetto ICMP di tipo *time-exceeded* con un codice *ttl-zero-during-transit* se questo avviene durante il transito sulla rete o *ttl-zero-during-reassembly* se questo avviene alla destinazione finale (vedi sez. B.3).

In sostanza grazie all'uso di questo accorgimento un pacchetto non può continuare a vagare indefinitamente sulla rete, e viene comunque scartato dopo un certo tempo, o meglio, dopo che ha attraversato in certo numero di router. Nel caso di Linux il valore iniziale utilizzato normalmente è 64 (vedi sez. 16.4.3).

B.1.3 Le opzioni di IP

Da fare ...

B.2 Il protocollo IPv6

Negli anni '90 con la crescita del numero di macchine connesse su Internet si arrivò a temere l'esaurimento dello spazio degli indirizzi disponibili, specie in vista di una prospettiva (per ora rivelatasi prematura) in cui ogni apparecchio elettronico sarebbe stato inserito all'interno della rete.

Per questo motivo si iniziò a progettare una nuova versione del protocollo

L'attuale Internet Protocol (IPv4) viene standardizzato nel 1981 dall'RFC 719; esso nasce per disaccoppiare le applicazioni della struttura hardware delle reti di trasmissione, e creare una interfaccia di trasmissione dei dati indipendente dal sottostante substrato di rete, che può essere realizzato con le tecnologie più disparate (Ethernet, Token Ring, FDDI, ecc.).

B.2.1 I motivi della transizione

Negli ultimi anni la crescita vertiginosa del numero di macchine connesse a internet ha iniziato a far emergere i vari limiti di IPv4; in particolare si è iniziata a delineare la possibilità di arrivare a una carenza di indirizzi disponibili.

In realtà il problema non è propriamente legato al numero di indirizzi disponibili; infatti con 32 bit si hanno 2^{32} , cioè circa 4 miliardi, numeri diversi possibili, che sono molti di più dei computer attualmente esistenti.

Il punto è che la suddivisione di questi numeri nei due livelli rete/host e l'utilizzo delle classi di indirizzamento mostrate in precedenza, ha comportato che, nella sua evoluzione storica, il dispiegamento delle reti e l'allocazione degli indirizzi siano stati inefficienti; neanche l'uso del CIDR ha permesso di eliminare le inefficienze che si erano formate, dato che il ridispiegamento degli indirizzi comporta cambiamenti complessi a tutti i livelli e la riassegnazione di tutti gli indirizzi dei computer di ogni sottorete.

Diventava perciò necessario progettare un nuovo protocollo che permettesse di risolvere questi problemi, e garantisse flessibilità sufficiente per poter continuare a funzionare a lungo termine; in particolare necessitava un nuovo schema di indirizzamento che potesse rispondere alle seguenti necessità:

- un maggior numero di numeri disponibili che consentisse di non restare più a corto di indirizzi
- un'organizzazione gerarchica più flessibile dell'attuale
- uno schema di assegnazione degli indirizzi in grado di minimizzare le dimensioni delle tabelle di instradamento
- uno spazio di indirizzi che consentisse un passaggio automatico dalle reti locali a internet

B.2.2 Principali caratteristiche di IPv6

Per rispondere alle esigenze descritte in sez. B.2.1 IPv6 nasce come evoluzione di IPv4, mantenendone inalterate le funzioni che si sono dimostrate valide, eliminando quelle inutili e aggiungendone poche altre ponendo al contempo una grande attenzione a mantenere il protocollo il più snello e veloce possibile.

I cambiamenti apportati sono comunque notevoli e possono essere riassunti a grandi linee nei seguenti punti:

- l'espansione delle capacità di indirizzamento e instradamento, per supportare una gerarchia con più livelli di indirizzamento, un numero di nodi indirizzabili molto maggiore e una auto-configurazione degli indirizzi
- l'introduzione un nuovo tipo di indirizzamento, l'*anycast* che si aggiungono agli usuali *unicast* e *multicast*
- la semplificazione del formato dell'intestazione, eliminando o rendendo opzionali alcuni dei campi di IPv4, per eliminare la necessità di riprocessare la stessa da parte dei router e contenere l'aumento di dimensione dovuto ai nuovi indirizzi
- un supporto per le opzioni migliorato, per garantire una trasmissione più efficiente del traffico normale, limiti meno stringenti sulle dimensioni delle opzioni, e la flessibilità necessaria per introdurne di nuove in futuro

- il supporto per delle capacità di qualità di servizio (QoS) che permetta di identificare gruppi di dati per i quali si può provvedere un trattamento speciale (in vista dell'uso di internet per applicazioni multimediali e/o “real-time”)

B.2.3 L'intestazione di IPv6

Per capire le caratteristiche di IPv6 partiamo dall'intestazione usata dal protocollo per gestire la trasmissione dei pacchetti; in fig. B.2 è riportato il formato dell'intestazione di IPv6 da confrontare con quella di IPv4 in fig. B.1. La spiegazione del significato dei vari campi delle due intestazioni è riportato rispettivamente in tab. B.5 e tab. B.3)

Figura B.2: L'intestazione o *header* di IPv6.

Come si può notare l'intestazione di IPv6 diventa di dimensione fissa, pari a 40 byte, contro una dimensione (minima, in assenza di opzioni) di 20 byte per IPv4; un semplice raddoppio nonostante lo spazio destinato agli indirizzi sia quadruplicato, questo grazie a una notevole semplificazione che ha ridotto il numero dei campi da 12 a 8.

Nome	Bit	Significato
<i>version</i>	4	La <i>versione</i> , nel caso specifico vale sempre 6.
<i>priority</i>	4	La <i>priorità</i> , vedi sez. B.2.15.
<i>flow label</i>	24	L' <i>etichetta di flusso</i> , vedi sez. B.2.14.
<i>payload length</i>	16	La <i>lunghezza del carico</i> , cioè del corpo dei dati che segue l'intestazione, in byte.
<i>next header</i>	8	L' <i>intestazione successiva</i> , identifica il tipo di pacchetto che segue l'intestazione di IPv6, ed usa gli stessi valori del campo protocollo nell'intestazione di IPv4.
<i>hop limit</i>	8	Il <i>limite di salti</i> , ha lo stesso significato del <i>time to live</i> nell'intestazione di IPv4.
<i>source IP</i>	128	L' <i>indirizzo di origine</i> .
<i>destination IP</i>	128	L' <i>indirizzo di destinazione</i> .

Tabella B.5: Legenda per il significato dei campi dell'intestazione di IPv6

Abbiamo già anticipato in sez. B.2.2 uno dei criteri principali nella progettazione di IPv6 è stato quello di ridurre al minimo il tempo di elaborazione dei pacchetti da parte dei router, un confronto con l'intestazione di IPv4 (vedi fig. B.1) mostra le seguenti differenze:

- è stato eliminato il campo *header length* in quanto le opzioni sono state tolte dall'intestazione che ha così dimensione fissa; ci possono essere più intestazioni opzionali (*intestazioni di estensione*, vedi sez. B.2.12), ciascuna delle quali avrà un suo campo di lunghezza all'interno.
- l'intestazione e gli indirizzi sono allineati a 64 bit, questo rende più veloce il processo da parte di computer con processori a 64 bit.
- i campi per gestire la frammentazione (*identification*, *flag* e *fragment offset*) sono stati eliminati; questo perché la frammentazione è un'eccezione che non deve rallentare l'elaborazione dei pacchetti nel caso normale.
- è stato eliminato il campo *checksum* in quanto tutti i protocolli di livello superiore (TCP, UDP e ICMPv6) hanno un campo di checksum che include, oltre alla loro intestazione e

ai dati, pure i campi *payload length*, *next header*, e gli indirizzi di origine e di destinazione; una checksum esiste anche per la gran parte protocolli di livello inferiore (anche se quelli che non lo hanno, come SLIP, non possono essere usati con grande affidabilità); con questa scelta si è ridotto di molto il tempo di elaborazione dato che i router non hanno più la necessità di ricalcolare la checksum ad ogni passaggio di un pacchetto per il cambiamento del campo *hop limit*.

- è stato eliminato il campo *type of service*, che praticamente non è mai stato utilizzato; una parte delle funzionalità ad esso delegate sono state reimplementate (vedi il campo *priority* al prossimo punto) con altri metodi.
- è stato introdotto un nuovo campo *flow label*, che viene usato, insieme al campo *priority* (che recupera i bit di precedenza del campo *type of service*) per implementare la gestione di una “qualità di servizio” (vedi sez. B.2.13) che permette di identificare i pacchetti appartenenti a un “flusso” di dati per i quali si può provvedere un trattamento speciale.

Oltre alle differenze precedenti, relative ai singoli campi nell'intestazione, ulteriori caratteristiche che diversificano il comportamento di IPv4 da quello di IPv6 sono le seguenti:

- il *broadcasting* non è previsto in IPv6, le applicazioni che lo usano devono essere reimplementate usando il *multicasting* (vedi sez. B.2.10), che da opzionale diventa obbligatorio.
- è stato introdotto un nuovo tipo di indirizzi, gli *anycast*.
- i router non possono più frammentare i pacchetti lungo il cammino, la frammentazione di pacchetti troppo grandi potrà essere gestita solo ai capi della comunicazione (usando un'apposita estensione vedi sez. B.2.12).
- IPv6 richiede il supporto per il *path MTU discovery* (cioè il protocollo per la selezione della massima lunghezza del pacchetto); seppure questo sia in teoria opzionale, senza di esso non sarà possibile inviare pacchetti più larghi della dimensione minima (576 byte).

B.2.4 Gli indirizzi di IPv6

Come già abbondantemente anticipato la principale novità di IPv6 è costituita dall'ampliamento dello spazio degli indirizzi, che consente di avere indirizzi disponibili in un numero dell'ordine di quello degli atomi che costituiscono la terra.

In realtà l'allocazione di questi indirizzi deve tenere conto della necessità di costruire delle gerarchie che consentano un instradamento rapido ed efficiente dei pacchetti, e flessibilità nel dispiegamento delle reti, il che comporta una riduzione drastica dei numeri utilizzabili; uno studio sull'efficienza dei vari sistemi di allocazione usati in altre architetture (come i sistemi telefonici) è comunque giunto alla conclusione che anche nella peggiore delle ipotesi IPv6 dovrebbe essere in grado di fornire più di un migliaio di indirizzi per ogni metro quadro della superficie terrestre.

B.2.5 La notazione

Con un numero di bit quadruplicato non è più possibile usare la notazione coi numeri decimali di IPv4 per rappresentare un numero IP. Per questo gli indirizzi di IPv6 sono in genere scritti come sequenze di otto numeri esadecimali di 4 cifre (cioè a gruppi di 16 bit) usando i due punti come separatore; cioè qualcosa del tipo 1080:0000:0000:0008:0800:ba98:2078:e3e3.

Visto che la notazione resta comunque piuttosto pesante esistono alcune abbreviazioni: si può evitare di scrivere gli zeri iniziali delle singole cifre, abbreviando l'indirizzo precedente in `1080:0:0:8:800:ba98:2078:e3e3`; se poi un intero è zero si può omettere del tutto, così come un insieme di zeri (ma questo solo una volta per non generare ambiguità) per cui il precedente indirizzo si può scrivere anche come `1080::8:800:ba98:2078:e3e3`.

Infine per scrivere un indirizzo IPv4 all'interno di un indirizzo IPv6 si può usare la vecchia notazione con i punti, per esempio `::192.84.145.138`.

Tipo di indirizzo	Prefisso	Frazione
riservato	0000 0000	1/256
non assegnato	0000 0001	1/256
riservato per NSAP	0000 001	1/128
riservato per IPX	0000 010	1/128
non assegnato	0000 011	1/128
non assegnato	0000 1	1/32
non assegnato	0001	1/16
provider-based	001	1/8
non assegnato	010	1/8
non assegnato	011	1/8
geografic-based	100	1/8
non assegnato	101	1/8
non assegnato	110	1/8
non assegnato	1110	1/16
non assegnato	1111 0	1/32
non assegnato	1111 10	1/64
non assegnato	1111 110	1/128
non assegnato	1111 1110 0	1/512
unicast link-local	1111 1110 10	1/1024
unicast site-local	1111 1110 11	1/1024
<i>multicast</i>	1111 1111	1/256

Tabella B.6: Classificazione degli indirizzi IPv6 a seconda dei bit più significativi

B.2.6 La architettura degli indirizzi di IPv6

Come per IPv4 gli indirizzi sono identificatori per una singola (indirizzi *unicast*) o per un insieme (indirizzi *multicast* e *anycast*) di interfacce di rete.

Gli indirizzi sono sempre assegnati all'interfaccia, non al nodo che la ospita; dato che ogni interfaccia appartiene ad un nodo quest'ultimo può essere identificato attraverso uno qualunque degli indirizzi unicast delle sue interfacce. A una interfaccia possono essere associati anche più indirizzi.

IPv6 presenta tre tipi diversi di indirizzi: due di questi, gli indirizzi *unicast* e *multicast* hanno le stesse caratteristiche che in IPv4, un terzo tipo, gli indirizzi *anycast* è completamente nuovo. In IPv6 non esistono più gli indirizzi *broadcast*, la funzione di questi ultimi deve essere reimplementata con gli indirizzi *multicast*.

Gli indirizzi *unicast* identificano una singola interfaccia: i pacchetti mandati ad un tale indirizzo verranno inviati a quella interfaccia, gli indirizzi *anycast* identificano un gruppo di interfacce tale che un pacchetto mandato a uno di questi indirizzi viene inviato alla più vicina (nel senso di distanza di routing) delle interfacce del gruppo, gli indirizzi *multicast* identificano un gruppo di interfacce tale che un pacchetto mandato a uno di questi indirizzi viene inviato a tutte le interfacce del gruppo.

In IPv6 non ci sono più le classi ma i bit più significativi indicano il tipo di indirizzo; in tab. B.6 sono riportati i valori di detti bit e il tipo di indirizzo che loro corrispondente.

I bit più significativi costituiscono quello che viene chiamato il *format prefix* ed è sulla base di questo che i vari tipi di indirizzi vengono identificati. Come si vede questa architettura di allocazione supporta l'allocazione di indirizzi per i provider, per uso locale e per il *multicast*; inoltre è stato riservato lo spazio per indirizzi NSAP, IPX e per le connessioni; gran parte dello spazio (più del 70%) è riservato per usi futuri.

Si noti infine che gli indirizzi *anycast* non sono riportati in tab. B.6 in quanto allocati al di fuori dello spazio di allocazione degli indirizzi unicast.

B.2.7 Indirizzi unicast *provider-based*

Gli indirizzi *provider-based* sono gli indirizzi usati per le comunicazioni globali, questi sono definiti nell'RFC 2073 e sono gli equivalenti degli attuali indirizzi delle classi da A a C.

L'autorità che presiede all'allocazione di questi indirizzi è la IANA; per evitare i problemi di crescita delle tabelle di instradamento e una procedura efficiente di allocazione la struttura di questi indirizzi è organizzata fin dall'inizio in maniera gerarchica; pertanto lo spazio di questi indirizzi è stato suddiviso in una serie di campi secondo lo schema riportato in tab. B.7.

3	5 bit	n bit	$56 - n$ bit	64 bit
010	Registry Id	Provider Id	Subscriber Id	Intra-Subscriber

Tabella B.7: Formato di un indirizzo unicast *provider-based*.

Al livello più alto la IANA può delegare l'allocazione a delle autorità regionali (i Regional Register) assegnando ad esse dei blocchi di indirizzi; a queste autorità regionali è assegnato un Registry Id che deve seguire immediatamente il prefisso di formato. Al momento sono definite tre registri regionali (INTERNIC, RIPE NCC e APNIC), inoltre la IANA si è riservata la possibilità di allocare indirizzi su base regionale; pertanto sono previsti i seguenti possibili valori per il *Registry Id*; gli altri valori restano riservati per la IANA.

Regione	Registro	Id
Nord America	INTERNIC	11000
Europa	RIPE NCC	01000
Asia	APNIC	00100
Multi-regionale	IANA	10000

Tabella B.8: Valori dell'identificativo dei Regional Register allocati ad oggi.

L'organizzazione degli indirizzi prevede poi che i due livelli successivi, di suddivisione fra *Provider Id*, che identifica i grandi fornitori di servizi, e *Subscriber Id*, che identifica i fruitori, sia gestita dai singoli registri regionali. Questi ultimi dovranno definire come dividere lo spazio di indirizzi assegnato a questi due campi (che ammonta a un totale di 56 bit), definendo lo spazio da assegnare al *Provider Id* e al *Subscriber Id*, ad essi spetterà inoltre anche l'allocazione dei numeri di *Provider Id* ai singoli fornitori, ai quali sarà delegata l'autorità di allocare i *Subscriber Id* al loro interno.

L'ultimo livello è quello *Intra-subscriber* che è lasciato alla gestione dei singoli fruitori finali, gli indirizzi *provider-based* lasciano normalmente gli ultimi 64 bit a disposizione per questo livello, la modalità più immediata è quella di usare uno schema del tipo mostrato in tab. B.9 dove l'*Interface Id* è dato dal MAC-address a 48 bit dello standard Ethernet, scritto in genere nell'hardware delle schede di rete, e si usano i restanti 16 bit per indicare la sottorete.

Qualora si dovesse avere a che fare con una necessità di un numero più elevato di sottoreti, il precedente schema andrebbe modificato, per evitare l'enorme spreco dovuto all'uso dei

64 bit	16 bit	48 bit
<i>Subscriber Prefix</i>	<i>Subnet Id</i>	<i>Interface Id</i>

Tabella B.9: Formato del campo *Intra-subscriber* per un indirizzo unicast *provider-based*.

MAC-address, a questo scopo si possono usare le capacità di auto-configurazione di IPv6 per assegnare indirizzi generici con ulteriori gerarchie per sfruttare efficacemente tutto lo spazio di indirizzi.

Un registro regionale può introdurre un ulteriore livello nella gerarchia degli indirizzi, allocando dei blocchi per i quali delegare l'autorità a dei registri nazionali, quest'ultimi poi avranno il compito di gestire la attribuzione degli indirizzi per i fornitori di servizi nell'ambito del/i paese coperto dal registro nazionale con le modalità viste in precedenza. Una tale ripartizione andrà effettuata all'interno dei soliti 56 bit come mostrato in tab. B.10.

3	5 bit	n bit	m bit	56-n-m bit	64 bit
3	<i>Reg.</i>	<i>Naz.</i>	<i>Prov.</i>	<i>Subscr.</i>	<i>Intra-Subscriber</i>

Tabella B.10: Formato di un indirizzo unicast *provider-based* che prevede un registro nazionale.

B.2.8 Indirizzi ad uso locale

Gli indirizzi ad uso locale sono indirizzi unicast che sono instradabili solo localmente (all'interno di un sito o di una sottorete), e possono avere una unicità locale o globale.

Questi indirizzi sono pensati per l'uso all'interno di un sito per mettere su una comunicazione locale immediata, o durante le fasi di auto-configurazione prima di avere un indirizzo globale.

10	54 bit	64 bit
FE80	0000 0000	<i>Interface Id</i>

Tabella B.11: Formato di un indirizzo *link-local*.

Ci sono due tipi di indirizzi, *link-local* e *site-local*. Il primo è usato per un singolo link; la struttura è mostrata in tab. B.11, questi indirizzi iniziano sempre con un valore nell'intervallo FE80–FEBF e vengono in genere usati per la configurazione automatica dell'indirizzo al bootstrap e per la ricerca dei vicini (vedi B.2.19); un pacchetto che abbia tale indirizzo come sorgente o destinazione non deve venire ritrasmesso dai router, sono gli indirizzi che identificano la macchina sulla rete locale, per questo sono chiamati in questo modo, in quanto sono usati solo su di essa.

Un indirizzo *site-local* invece è usato per l'indirizzamento all'interno di un sito che non necessita di un prefisso globale; la struttura è mostrata in tab. B.12, questi indirizzi iniziano sempre con un valore nell'intervallo FEC0–FEFF e non devono venire ritrasmessi dai router all'esterno del sito stesso; sono in sostanza gli equivalenti degli indirizzi riservati per reti private definiti su IPv4. Per entrambi gli indirizzi il campo *Interface Id* è un identificatore che deve essere unico nel dominio in cui viene usato, un modo immediato per costruirlo è quello di usare il MAC-address delle schede di rete.

Gli indirizzi di uso locale consentono ad una organizzazione che non è (ancora) connessa ad Internet di operare senza richiedere un prefisso globale, una volta che in seguito l'organizzazione venisse connessa a Internet potrebbe continuare a usare la stessa suddivisione effettuata

10	38 bit	16 bit	64 bit
FEC0	0000 . . . 0000	Subnet Id	Interface Id

Tabella B.12: Formato di un indirizzo *site-local*.

con gli indirizzi *site-local* utilizzando un prefisso globale e la rinumerazione degli indirizzi delle singole macchine sarebbe automatica.

B.2.9 Indirizzi riservati

Alcuni indirizzi sono riservati per scopi speciali, in particolare per scopi di compatibilità.

Un primo tipo sono gli indirizzi *IPv4 mappati su IPv6* (mostrati in tab. B.13), questo sono indirizzi unicast che vengono usati per consentire ad applicazioni IPv6 di comunicare con host capaci solo di IPv4; questi sono ad esempio gli indirizzi generati da un DNS quando l'host richiesto supporta solo IPv4; l'uso di un tale indirizzo in un socket IPv6 comporta la generazione di un pacchetto IPv4 (ovviamente occorre che sia IPv4 che IPv6 siano supportati sull'host di origine).

80 bit	16 bit	32 bit
0000 0000	FFFF	IPv4 address

Tabella B.13: Formato di un indirizzo IPV4 mappato su IPv6.

Un secondo tipo di indirizzi di compatibilità sono gli *IPv4 compatibili IPv6* (vedi tab. B.14) usati nella transizione da IPv4 a IPv6: quando un nodo che supporta sia IPv6 che IPv4 non ha un router IPv6 deve usare nel DNS un indirizzo di questo tipo, ogni pacchetto IPv6 inviato a un tale indirizzo verrà automaticamente incapsulato in IPv4.

80 bit	16 bit	32 bit
0000 0000	0000	IPv4 address

Tabella B.14: Formato di un indirizzo IPV4 mappato su IPv6.

Altri indirizzi speciali sono il *loopback address*, costituito da 127 zeri ed un uno (cioè ::1) e l'*indirizzo generico* costituito da tutti zeri (scritto come 0::0 o ancora più semplicemente come :) usato in genere quando si vuole indicare l'accettazione di una connessione da qualunque host.

B.2.10 Multicasting

Gli indirizzi *multicast* sono usati per inviare un pacchetto a un gruppo di interfacce; l'indirizzo identifica uno specifico gruppo di *multicast* e il pacchetto viene inviato a tutte le interfacce di detto gruppo. Un'interfaccia può appartenere ad un numero qualunque numero di gruppi di *multicast*. Il formato degli indirizzi *multicast* è riportato in tab. B.15:

Il prefisso di formato per tutti gli indirizzi *multicast* è FF, ad esso seguono i due campi il cui significato è il seguente:

- *flag*: un insieme di 4 bit, di cui i primi tre sono riservati e posti a zero, l'ultimo è zero se l'indirizzo è permanente (cioè un indirizzo noto, assegnato dalla IANA), ed è uno se invece l'indirizzo è transitorio.

8	4	4	112 bit
FF	flag	scop	Group Id

Tabella B.15: Formato di un indirizzo *multicast*.

- *scop* è un numero di quattro bit che indica il raggio di validità dell'indirizzo, i valori assegnati per ora sono riportati in tab. B.16.

Gruppi di multicast			
0	riservato	8	organizzazione locale
1	nodo locale	9	non assegnato
2	collegamento locale	A	non assegnato
3	non assegnato	B	non assegnato
4	non assegnato	C	non assegnato
5	sito locale	D	non assegnato
6	non assegnato	E	globale
7	non assegnato	F	riservato

Tabella B.16: Possibili valori del campo *scop* di un indirizzo *multicast*.

Infine l'ultimo campo identifica il gruppo di *multicast*, sia permanente che transitorio, all'interno del raggio di validità del medesimo. Alcuni indirizzi *multicast*, riportati in tab. B.17 sono già riservati per il funzionamento della rete.

Uso	Indirizzi riservati	Definizione
all-nodes	FFxx:0:0:0:0:0:1	RFC 1970
all-routers	FFxx:0:0:0:0:0:2	RFC 1970
all-rip-routers	FFxx:0:0:0:0:0:9	RFC 2080
all-cbt-routers	FFxx:0:0:0:0:0:10	
reserved	FFxx:0:0:0:0:0:1:0	IANA
link-name	FFxx:0:0:0:0:0:1:1	
all-dhcp-agents	FFxx:0:0:0:0:0:1:2	
all-dhcp-servers	FFxx:0:0:0:0:0:1:3	
all-dhcp-relays	FFxx:0:0:0:0:0:1:4	
solicited-nodes	FFxx:0:0:0:0:1:0:0	RFC 1970

Tabella B.17: Gruppi di *multicast* predefiniti.

L'utilizzo del campo di *scope* e di questi indirizzi predefiniti serve a recuperare le funzionalità del *broadcasting* (ad esempio inviando un pacchetto all'indirizzo FF02:0:0:0:0:0:1 si raggiungono tutti i nodi locali).

B.2.11 Indirizzi *anycast*

Gli indirizzi *anycast* sono indirizzi che vengono assegnati ad un gruppo di interfacce: un pacchetto indirizzato a questo tipo di indirizzo viene inviato al componente del gruppo più "vicino" secondo la distanza di instradamento calcolata dai router.

Questi indirizzi sono allocati nello stesso spazio degli indirizzi unicast, usando uno dei formati disponibili, e per questo, sono da essi assolutamente indistinguibili. Quando un indirizzo unicast viene assegnato a più interfacce (trasformandolo in un *anycast*) il computer su cui è l'interfaccia deve essere configurato per tener conto del fatto.

Gli indirizzi *anycast* consentono a un nodo sorgente di inviare pacchetti a una destinazione su un gruppo di possibili interfacce selezionate. La sorgente non deve curarsi di come scegliere

l'interfaccia più vicina, compito che tocca al sistema di instradamento (in sostanza la sorgente non ha nessun controllo sulla selezione).

Gli indirizzi anycast, quando vengono usati come parte di una sequenza di instradamento, consentono ad esempio ad un nodo di scegliere quale fornitore vuole usare (configurando gli indirizzi anycast per identificare i router di uno stesso provider).

Questi indirizzi pertanto possono essere usati come indirizzi intermedi in una intestazione di instradamento o per identificare insiemi di router connessi a una particolare sottorete, o che forniscono l'accesso a un certo sotto dominio.

L'idea alla base degli indirizzi anycast è perciò quella di utilizzarli per poter raggiungere il fornitore di servizio più vicino; ma restano aperte tutta una serie di problematiche, visto che una connessione con uno di questi indirizzi non è possibile, dato che per una variazione delle distanze di routing non è detto che due pacchetti successivi finiscano alla stessa interfaccia.

La materia è pertanto ancora controversa e in via di definizione.

B.2.12 Le estensioni

Come già detto in precedenza IPv6 ha completamente cambiato il trattamento delle opzioni; queste ultime infatti sono state tolte dall'intestazione del pacchetto, e poste in apposite *intestazioni di estensione* (o *extension header*) poste fra l'intestazione di IPv6 e l'intestazione del protocollo di trasporto.

Per aumentare la velocità di elaborazione, sia dei dati del livello seguente che di ulteriori opzioni, ciascuna estensione deve avere una lunghezza multipla di 8 byte per mantenere l'allineamento a 64 bit di tutti le intestazioni seguenti.

Dato che la maggior parte di queste estensioni non sono esaminate dai router durante l'instradamento e la trasmissione dei pacchetti, ma solo all'arrivo alla destinazione finale, questa scelta ha consentito un miglioramento delle prestazioni rispetto a IPv4 dove la presenza di un'opzione comportava l'esame di tutte quante.

Un secondo miglioramento è che rispetto a IPv4 le opzioni possono essere di lunghezza arbitraria e non limitate a 40 byte; questo, insieme al modo in cui vengono trattate, consente di utilizzarle per scopi come l'autenticazione e la sicurezza, improponibili con IPv4.

Le estensioni definite al momento sono le seguenti:

- **Hop by hop** devono seguire immediatamente l'intestazione principale; indicano le opzioni che devono venire processate ad ogni passaggio da un router, fra di esse è da menzionare la *jumbo payload* che segnala la presenza di un pacchetto di dati di dimensione superiore a 65535 byte.
- **Destination options** opzioni che devono venire esaminate al nodo di ricevimento, nessuna di esse è tuttora definita.
- **Routing** definisce una *source route* (come la analoga opzione di IPv4) cioè una lista di indirizzi IP di nodi per i quali il pacchetto deve passare.
- **Fragmentation** viene generato automaticamente quando un host vuole frammentare un pacchetto, ed è riprocessato automaticamente alla destinazione che riassembla i frammenti.
- **Authentication** gestisce l'autenticazione e il controllo di integrità dei pacchetti; è documentato dall'RFC 1826.
- **Encapsulation** serve a gestire la segretezza del contenuto trasmesso; è documentato dall'RFC 1827.

La presenza di opzioni è rilevata dal valore del campo *next header* che indica qual è l'intestazione successiva a quella di IPv6; in assenza di opzioni questa sarà l'intestazione di un protocollo di trasporto del livello superiore, per cui il campo assumerà lo stesso valore del campo *protocol* di IPv4, altrimenti assumerà il valore dell'opzione presente; i valori possibili sono riportati in tab. B.18.

Valore	Keyword	Tipo di protocollo
0		Riservato.
	HBH	Hop by Hop.
1	ICMP	Internet Control Message (IPv4 o IPv6).
2	IGMP	Internet Group Management (IPv4).
3	GGP	Gateway-to-Gateway.
4	IP	IP in IP (IPv4 encapsulation).
5	ST	Stream.
6	TCP	Trasmission Control.
17	UDP	User Datagram.
43	RH	Routing Header (IPv6).
44	FH	Fragment Header (IPv6).
45	IDRP	Inter Domain Routing.
51	AH	Authentication Header (IPv6).
52	ESP	Encrypted Security Payload (IPv6).
59	Null	No next header (IPv6).
88	IGRP	Internet Group Routing.
89	OSPF	Open Short Path First.
255		Riservato.

Tabella B.18: Tipi di protocolli e intestazioni di estensione

Questo meccanismo permette la presenza di più opzioni in successione prima del pacchetto del protocollo di trasporto; l'ordine raccomandato per le estensioni è quello riportato nell'elenco precedente con la sola differenza che le opzioni di destinazione sono inserite nella posizione ivi indicata solo se, come per il tunnelling, devono essere esaminate dai router, quelle che devono essere esaminate solo alla destinazione finale vanno in coda.

B.2.13 Qualità di servizio

Una delle caratteristiche innovative di IPv6 è quella di avere introdotto un supporto per la qualità di servizio che è importante per applicazioni come quelle multimediali o “real-time” che richiedono un qualche grado di controllo sulla stabilità della banda di trasmissione, sui ritardi o la dispersione del tempo del flusso dei pacchetti.

B.2.14 Etichette di flusso

L'introduzione del campo *flow label* può essere usata dall'origine della comunicazione per etichettare quei pacchetti per i quali si vuole un trattamento speciale da parte dei router come un una garanzia di banda minima assicurata o un tempo minimo di instradamento/trasmissione garantito.

Questo aspetto di IPv6 è ancora sperimentale per cui i router che non supportino queste funzioni devono porre a zero il *flow label* per i pacchetti da loro originanti e lasciare invariato il campo per quelli in transito.

Un flusso è una sequenza di pacchetti da una particolare origine a una particolare destinazione per il quale l'origine desidera un trattamento speciale da parte dei router che lo

manipolano; la natura di questo trattamento può essere comunicata ai router in vari modi (come un protocollo di controllo o con opzioni del tipo *hop-by-hop*).

Ci possono essere più flussi attivi fra un'origine e una destinazione, come del traffico non assegnato a nessun flusso, un flusso viene identificato univocamente dagli indirizzi di origine e destinazione e da una etichetta di flusso diversa da zero, il traffico normale deve avere l'etichetta di flusso posta a zero.

L'etichetta di flusso è assegnata dal nodo di origine, i valori devono essere scelti in maniera (pseudo)casuale nel range fra 1 e FFFFFFF in modo da rendere utilizzabile un qualunque sottoinsieme dei bit come chiavi di hash per i router.

B.2.15 Priorità

Il campo di priorità consente di indicare il livello di priorità dei pacchetti relativamente agli altri pacchetti provenienti dalla stessa sorgente. I valori sono divisi in due intervalli, i valori da 0 a 7 sono usati per specificare la priorità del traffico per il quale la sorgente provvede un controllo di congestione cioè per il traffico che può essere “tirato indietro” in caso di congestione come quello di TCP, i valori da 8 a 15 sono usati per i pacchetti che non hanno questa caratteristica, come i pacchetti “real-time” inviati a ritmo costante.

Per il traffico con controllo di congestione sono raccomandati i seguenti valori di priorità a seconda del tipo di applicazione:

Valore	Tipo di traffico
0	Traffico generico.
1	Traffico di riempimento (es. news).
2	Trasferimento dati non interattivo (es. e-mail).
3	Riservato.
4	Trasferimento dati interattivo (es. FTP, HTTP, NFS).
5	Riservato.

Tabella B.19: Formato di un indirizzo *site-local*.

Per il traffico senza controllo di congestione la priorità più bassa dovrebbe essere usata per quei pacchetti che si preferisce siano scartati più facilmente in caso di congestione.

B.2.16 Sicurezza a livello IP

La attuale implementazione di Internet presenta numerosi problemi di sicurezza, in particolare i dati presenti nelle intestazioni dei vari protocolli sono assunti essere corretti, il che dà adito alla possibilità di varie tipologie di attacco forgiando pacchetti false, inoltre tutti questi dati passano in chiaro sulla rete e sono esposti all'osservazione di chiunque si trovi in mezzo.

Con IPv4 non è possibile realizzare un meccanismo di autenticazione e riservatezza a un livello inferiore al primo (quello di applicazione), con IPv6 è stata progettata la possibilità di intervenire al livello di rete (il terzo) prevedendo due apposite estensioni che possono essere usate per fornire livelli di sicurezza a seconda degli utenti. La codifica generale di questa architettura è riportata nell'RFC 2401.

Il meccanismo in sostanza si basa su due estensioni:

- una intestazione di sicurezza (*authentication header*) che garantisce al destinatario l'autenticità del pacchetto
- un carico di sicurezza (*Encrypted Security Payload*) che assicura che solo il legittimo ricevente può leggere il pacchetto.

Perché tutto questo funzioni le stazioni sorgente e destinazione devono usare una stessa chiave crittografica e gli stessi algoritmi, l'insieme degli accordi fra le due stazioni per concordare chiavi e algoritmi usati va sotto il nome di associazione di sicurezza.

I pacchetti autenticati e crittografati portano un indice dei parametri di sicurezza (SPI, *Security Parameter Index*) che viene negoziato prima di ogni comunicazione ed è definito dalla stazione sorgente. Nel caso di *multicast* dovrà essere lo stesso per tutte le stazioni del gruppo.

B.2.17 Autenticazione

Il primo meccanismo di sicurezza è quello dell'intestazione di autenticazione (*authentication header*) che fornisce l'autenticazione e il controllo di integrità (ma senza riservatezza) dei pacchetti IP.

L'intestazione di autenticazione ha il formato descritto in fig. B.3: il campo *Next Header* indica l'intestazione successiva, con gli stessi valori del campo omonimo nell'intestazione principale di IPv6, il campo *Length* indica la lunghezza dell'intestazione di autenticazione in numero di parole a 32 bit, il campo riservato deve essere posto a zero, seguono poi l'indice di sicurezza, stabilito nella associazione di sicurezza, e un numero di sequenza che la stazione sorgente deve incrementare di pacchetto in pacchetto.

Completano l'intestazione i dati di autenticazione che contengono un valore di controllo di integrità (ICV, *Integrity Check Value*), che deve essere di dimensione pari a un multiplo intero di 32 bit e può contenere un padding per allineare l'intestazione a 64 bit. Tutti gli algoritmi di autenticazione devono provvedere questa capacità.

Figura B.3: Formato dell'intestazione dell'estensione di autenticazione.

L'intestazione di autenticazione può essere impiegata in due modi diverse modalità: modalità trasporto e modalità tunnel.

La modalità trasporto è utilizzabile solo per comunicazioni fra stazioni singole che supportino l'autenticazione. In questo caso l'intestazione di autenticazione è inserita dopo tutte le altre intestazioni di estensione eccezion fatta per la *Destination Option* che può comparire sia prima che dopo.

Figura B.4: Formato di un pacchetto IPv6 che usa l'opzione di autenticazione.

La modalità tunnel può essere utilizzata sia per comunicazioni fra stazioni singole che con un gateway di sicurezza; in questa modalità ...

L'intestazione di autenticazione è una intestazione di estensione inserita dopo l'intestazione principale e prima del carico dei dati. La sua presenza non ha perciò alcuna influenza sui livelli superiori dei protocolli di trasmissione come il TCP.

La procedura di autenticazione cerca di garantire l'autenticità del pacchetto nella massima estensione possibile, ma dato che alcuni campi dell'intestazione di IP possono variare in maniera imprevedibile alla sorgente, il loro valore non può essere protetto dall'autenticazione.

Il calcolo dei dati di autenticazione viene effettuato alla sorgente su una versione speciale del pacchetto in cui il numero di salti nell'intestazione principale è impostato a zero, così come le opzioni che possono essere modificate nella trasmissione, e l'intestazione di routing (se usata) è posta ai valori che deve avere all'arrivo.

L'estensione è indipendente dall'algoritmo particolare, e il protocollo è ancora in fase di definizione; attualmente è stato suggerito l'uso di una modifica dell'MD5 chiamata *keyed MD5*

che combina alla codifica anche una chiave che viene inserita all'inizio e alla fine degli altri campi.

B.2.18 Riservatezza

Per garantire una trasmissione riservata dei dati, è stata prevista la possibilità di trasmettere pacchetti con i dati criptati: il cosiddetto ESP, *Encrypted Security Payload*. Questo viene realizzato usando con una apposita opzione che deve essere sempre l'ultima delle intestazioni di estensione; ad essa segue il carico del pacchetto che viene criptato.

Un pacchetto crittografato pertanto viene ad avere una struttura del tipo di quella mostrata in fig. B.5, tutti i campi sono in chiaro fino al vettore di inizializzazione, il resto è crittografato.

Figura B.5: Schema di pacchetto crittografato.

B.2.19 Auto-configurazione

Una delle caratteristiche salienti di IPv6 è quella dell'auto-configurazione, il protocollo infatti fornisce la possibilità ad un nodo di scoprire automaticamente il suo indirizzo acquisendo i parametri necessari per potersi connettere a internet.

L'auto-configurazione sfrutta gli indirizzi *link-local*; qualora sul nodo sia presente una scheda di rete che supporta lo standard IEEE802 (ethernet) questo garantisce la presenza di un indirizzo fisico a 48 bit unico; pertanto il nodo può assumere automaticamente senza pericoli di collisione l'indirizzo *link-local* FE80::xxxx:xxxx:xxxx dove xxxx:xxxx:xxxx è l'indirizzo hardware della scheda di rete.

Nel caso in cui non sia presente una scheda che supporta lo standard IEEE802 allora il nodo assumerà ugualmente un indirizzo *link-local* della forma precedente, ma il valore di xxxx:xxxx:xxxx sarà generato casualmente; in questo caso la probabilità di collisione è di 1 su 300 milioni. In ogni caso per prevenire questo rischio il nodo invierà un messaggio ICMP *Solicitation* all'indirizzo scelto attendendo un certo lasso di tempo; in caso di risposta l'indirizzo è duplicato e il procedimento dovrà essere ripetuto con un nuovo indirizzo (o interrotto richiedendo assistenza).

Una volta ottenuto un indirizzo locale valido diventa possibile per il nodo comunicare con la rete locale; sono pertanto previste due modalità di auto-configurazione, descritte nelle seguenti sezioni. In ogni caso l'indirizzo *link-local* resta valido.

B.2.20 Auto-configurazione stateless

Questa è la forma più semplice di auto-configurazione, possibile quando l'indirizzo globale può essere ricavato dall'indirizzo *link-local* cambiando semplicemente il prefisso a quello assegnato dal provider per ottenere un indirizzo globale.

La procedura di configurazione è la seguente: all'avvio tutti i nodi IPv6 iniziano si devono aggregare al gruppo di *multicast all-nodes* programmando la propria interfaccia per ricevere i messaggi dall'indirizzo *multicast* FF02::1 (vedi sez. B.2.10); a questo punto devono inviare un messaggio ICMP *Router solicitation* a tutti i router locali usando l'indirizzo *multicast* FF02::2 usando come sorgente il proprio indirizzo *link-local*.

Il router risponderà con un messaggio ICMP *Router Advertisement* che fornisce il prefisso e la validità nel tempo del medesimo, questo tipo di messaggio può essere trasmesso anche a

intervalli regolari. Il messaggio contiene anche l'informazione che autorizza un nodo a auto-costruire l'indirizzo, nel qual caso, se il prefisso unito all'indirizzo *link-local* non supera i 128 bit, la stazione ottiene automaticamente il suo indirizzo globale.

B.2.21 Auto-configurazione stateful

Benché estremamente semplice l'auto-configurazione stateless presenta alcuni problemi; il primo è che l'uso degli indirizzi delle schede di rete è molto inefficiente; nel caso in cui ci siano esigenze di creare una gerarchia strutturata su parecchi livelli possono non restare 48 bit per l'indirizzo della singola stazione; il secondo problema è di sicurezza, dato che basta introdurre in una rete una stazione autoconfigurante per ottenere un accesso legale.

Per questi motivi è previsto anche un protocollo stateful basato su un server che offra una versione IPv6 del DHCP; un apposito gruppo di *multicast* FF02::1:0 è stato riservato per questi server; in questo caso il nodo interrogherà il server su questo indirizzo di *multicast* con l'indirizzo *link-local* e riceverà un indirizzo unicast globale.

B.3 Il protocollo ICMP

Come già accennato nelle sezioni precedenti, l'*Internet Control Message Protocol* è un protocollo di servizio fondamentale per il funzionamento del livello di rete. Il protocollo ICMP viene trasportato direttamente su IP, ma proprio per questa sua caratteristica di protocollo di servizio è da considerarsi a tutti gli effetti appartenente al livello di rete.

B.3.1 L'intestazione di ICMP

Il protocollo ICMP è estremamente semplice, ed il suo unico scopo è quello di inviare messaggi di controllo; in fig. B.6 si è riportata la struttura dell'intestazione di un pacchetto ICMP generico.

Figura B.6: L'intestazione del protocollo ICMP.

Ciascun pacchetto ICMP è contraddistinto dal valore del primo campo, il tipo, che indica appunto che tipo di messaggio di controllo viene veicolato dal pacchetto in questione; i valori possibili per questo campo, insieme al relativo significato, sono riportati in tab. B.20.

Per alcuni tipi di messaggi ICMP, esiste un secondo campo, detto codice, che specifica ulteriormente la natura del messaggio; i soli messaggi che utilizzano un valore per questo campo sono quelli di tipo *destination-unreachable*, *redirect*, *time-exceeded* e *parameter-problem*. I possibili valori del codice relativi a ciascuno di essi sono stati riportati nelle quattro sezioni in cui si è suddivisa tab. B.21, rispettivamente nell'ordine con cui sono appena elencati i tipi a cui essi fanno riferimento.

Valore	Tipo	Significato
any	–	Seleziona tutti i possibili valori
<i>echo-reply</i>	0	Inviato in risposta ad un ICMP <i>echo-request</i> .
<i>destination-unreachable</i>	3	Segnala una destinazione irraggiungibile, viene inviato all'IP sorgente di un pacchetto quando un router realizza che questo non può essere inviato a destinazione.
<i>source-quench</i>	4	Inviato in caso di congestione della rete per indicare all'IP sorgente di diminuire il traffico inviato.
<i>redirect</i>	5	Inviato per segnalare un errore di routing, richiede che la macchina sorgente reindiriga il traffico ad un altro router da esso specificato.
<i>echo-request</i>	8	Richiede l'invio in risposta di un <i>echo-reply</i> .
<i>time-exceeded</i>	11	Inviato quando il TTL di un pacchetto viene azzerato.
<i>parameter-problem</i>	12	Inviato da un router che rileva dei problemi con l'intestazione di un pacchetto.
<i>timestamp-request</i>	13	Richiede l'invio in risposta di un <i>timestamp-reply</i> .
<i>timestamp-reply</i>	14	Inviato in risposta di un <i>timestamp-request</i> .
<i>info-request</i>	15	Richiede l'invio in risposta di un <i>info-reply</i> .
<i>info-reply</i>	16	Inviato in risposta di un <i>info-request</i> .
<i>address-mask-request</i>	17	Richiede l'invio in risposta di un <i>address-mask-reply</i> .
<i>address-mask-reply</i>	18	Inviato in risposta di un <i>address-mask-request</i> .

Tabella B.20: I valori del tipo per i pacchetti ICMP.

Valore	Codice
<i>network-unreachable</i>	0
<i>host-unreachable</i>	1
<i>protocol-unreachable</i>	2
<i>port-unreachable</i>	3
<i>fragmentation-needed</i>	4
<i>source-route-failed</i>	5
<i>network-unknown</i>	6
<i>host-unknown</i>	7
<i>host-isolated</i>	8
<i>network-prohibited</i>	9
<i>host-prohibited</i>	10
<i>TOS-network-unreachable</i>	11
<i>TOS-host-unreachable</i>	12
<i>communication-prohibited</i>	13
<i>host-precedence-violation</i>	14
<i>precedence-cutoff</i>	15
<i>network-redirect</i>	0
<i>host-redirect</i>	1
<i>TOS-network-redirect</i>	2
<i>TOS-host-redirect</i>	3
<i>ttl-zero-during-transit</i>	0
<i>ttl-zero-during-reassembly</i>	1
<i>ip-header-bad</i>	0
<i>required-option-missing</i>	1

Tabella B.21: Valori del campo codice per il protocollo ICMP.

Appendice C

Il livello di trasporto

In questa appendice tratteremo i vari protocolli relativi al livello di trasporto.¹ In particolare gran parte del capitolo sarà dedicato al più importante di questi, il TCP, che è pure il più complesso ed utilizzato su internet.

C.1 Il protocollo TCP

In questa sezione prenderemo in esame i vari aspetti del protocollo TCP, il protocollo più comunemente usato dalle applicazioni di rete.

C.1.1 Gli stati del TCP

In sez. 15.1 abbiamo descritto in dettaglio le modalità con cui il protocollo TCP avvia e conclude una connessione, ed abbiamo accennato alla presenza dei vari stati del protocollo. In generale infatti il funzionamento del protocollo segue una serie di regole, che possono essere riassunte nel comportamento di una macchina a stati, il cui diagramma di transizione è riportato in fig. C.1.

Figura C.1: Il diagramma degli stati del TCP.

Il protocollo prevede l'esistenza di 11 diversi stati per una connessione ed un insieme di regole per le transizioni da uno stato all'altro basate sullo stato corrente, sull'operazione effettuata dall'applicazione o sul tipo di segmento ricevuto; i nomi degli stati mostrati in fig. C.1 sono gli stessi che vengono riportati del comando `netstat` nel campo *State*.

Figura C.2: L'intestazione del protocollo TCP.

C.2 Il protocollo UDP

In questa sezione prenderemo in esame i vari aspetti del protocollo UDP, che dopo il TCP è il protocollo più usato dalle applicazioni di rete.

¹al solito per la definizione dei livelli si faccia riferimento alle spiegazioni fornite in sez. 13.2.

Figura C.3: L'intestazione del protocollo UDP.

Appendice D

Gli strumenti di ausilio per la programmazione

Tratteremo in questa appendice in maniera superficiale i principali strumenti che vengono utilizzati per programmare in ambito Linux, ed in particolare gli strumenti per la compilazione e la costruzione di programmi e librerie, e gli strumenti di gestione dei sorgenti e di controllo di versione.

Questo materiale è ripreso da un vecchio articolo, ed al momento è molto obsoleto.

D.1 L'uso di `make` per l'automazione della compilazione

Il comando `make` serve per automatizzare il processo di costruzione di un programma ed effettuare una compilazione intelligente di tutti i file relativi ad un progetto software, ricompilando solo i file necessari ed eseguendo automaticamente tutte le operazioni che possono essere necessarie alla produzione del risultato finale.¹

D.1.1 Introduzione a `make`

Con `make` si possono definire i simboli del preprocessore C che consentono la compilazione condizionale dei programmi (anche in Fortran); è pertanto possibile gestire la ricompilazione dei programmi con diverse configurazioni con la modifica di un unico file.

La sintassi normale del comando (quella che si usa quasi sempre, per le opzioni vedere la pagina di manuale) è semplicemente `make`. Questo comando esegue le istruzioni contenute in un file standard (usualmente `Makefile`, o `makefile` nella directory corrente).

Il formato normale dei comandi contenuti in un `Makefile` è:

```
bersaglio: dipendenza1 dipendenza2 ...
    regola1
    regola2
    ...
```

dove lo spazio all'inizio deve essere un tabulatore (metterci degli spazi è un errore comune, fortunatamente ben segnalato dalle ultime versioni del programma), il bersaglio e le dipendenze nomi di file e le regole comandi di shell.

¹in realtà `make` non si applica solo ai programmi, ma in generale alla automazione di processi di costruzione, ad esempio anche la creazione dei file di questa guida viene fatta con `make`.

Il concetto di base è che se uno dei file di dipendenza è più recente (nel senso di tempo di ultima modifica) del file bersaglio quest'ultimo viene ricostruito di nuovo usando le regole elencate nelle righe successive.

Il comando `make` ricostruisce di default il primo bersaglio che viene trovato nella scansione del `Makefile`, se in un `Makefile` sono contenuti più bersagli indipendenti, si può farne ricostruire un altro che non sia il primo passandolo esplicitamente al comando come argomento, con qualcosa del tipo di: `make altrobersaglio`.

Si tenga presente che le dipendenze stesse possono essere dichiarate come bersagli dipendenti da altri file; in questo modo è possibile creare una catena di ricostruzioni.

In esempio comune di quello che si fa è mettere come primo bersaglio il programma principale che si vuole usare, e come dipendenze tutte gli oggetti delle funzioni subordinate che utilizza, con i quali deve essere collegato; a loro volta questi oggetti sono bersagli che hanno come dipendenza i relativi sorgenti. In questo modo il cambiamento di una delle funzioni subordinate comporta solo la ricompilazione della medesima e del programma finale.

D.1.2 Utilizzo di make

Il comando `make` mette a disposizione una serie molto complesse di opzioni e di regole standard predefinite e sottintese, che permettono una gestione estremamente rapida e concisa di progetti anche molto complessi; per questo piuttosto che fare una replica del manuale preferisco commentare un esempio di `makefile`, quello usato per ricompilare i programmi di analisi dei dati dei test su fascio del tracciatore di Pamela.

```
#-----
#
# Makefile for a Linux System:
# use GNU FORTRAN compiler g77
# Makefile done for tracker test data
#
#-----
# Fortran flags
FC=g77
FFLAGS= -fvxt -fno-automatic -Wall -O6 -DPC # -DDEBUG
CC=gcc
CFLAGS= -Wall -O6
CFLADJ=-c #-DDEBUG
#
# FC          Fortran compiler for standard rules
# FFLAGS      Fortran flags for standard rules
# CC          C Compiler for standard rules
# CFLAGS      C compiler flags for standard rules
LIBS= -L/cern/pro/lib -lkernlib -lpacklib -lgrafflib -lmathlib
OBJ=cnoise.o fit2.o pedsig.o loop.o badstrp.o cutcn.o readevnt.o \
erasepedvar.o readinit.o dumpval.o writeinit.o

riduzione: riduzione.F $(OBJ) commondef.f readfile.o
        $(FC) $(FFLAGS) -o riduzione riduzione.F readfile.o $(OBJ) $(LIBS)

readfile.o: readfile.c
        $(CC) $(CFLAGS) -o readfile.o readfile.c

$(OBJ): commondef.f
```

```
.PHONY : clean
clean:
    rm -f *.o
    rm -f *~
    rm -f riduzione
    rm -f *.rz
    rm -f output
```

Anzitutto i commenti, ogni linea che inizia con un # è un commento e non viene presa in considerazione.

Con `make` possono essere definite delle variabili, da potersi riusare a piacimento, per leggibilità si tende a definirle tutte maiuscole, nell'esempio ne sono definite varie:

```
FC=g77
FFLAGS= -fvxt -fno-automatic -Wall -O6 -DPC # -DDEBUG
CC=gcc
CFLAGS= -Wall -O6
CFLADJ=-c #-DDEBUG
...
LIBS= -L/cern/pro/lib -lkernlib -lpacklib -lgrafflib -lmathlib
OBJ=cnoise.o fit2.o pedsig.o loop.o badstrp.o cutcn.o readevnt.o \
```

La sintassi è `NOME=`, alcuni nomi però hanno un significato speciale (nel caso `FC`, `FLAGS`, `CC`, `CFLAGS`) in quanto sono usati da `make` nelle cosiddette *regole implicite* (su cui torneremo dopo).

Nel caso specifico, vedi anche i commenti, abbiamo definito i comandi di compilazione da usare per il C e il Fortran, e i rispettivi flag, una variabile che contiene il pathname e la lista delle librerie del CERN e una variabile con una lista di file oggetto.

Per richiamare una variabile si usa la sintassi `$(NOME)`, ad esempio nel `makefile` abbiamo usato:

```
$(FC) $(FFLAGS) -o riduzione riduzione.F readfile.o $(OBJ) $(LIBS)
```

e questo significa che la regola verrà trattata come se avessimo scritto esplicitamente i valori delle variabili.

Veniamo ora alla parte principale del `makefile` che esegue la costruzione del programma:

```
riduzione: riduzione.F $(OBJ) commondef.f readfile.o
    $(FC) $(FFLAGS) -o riduzione riduzione.F readfile.o $(OBJ) $(LIBS)
readfile.o: readfile.c
    $(CC) $(CFLAGS) -o readfile.o readfile.c
$(OBJ): commondef.f
```

Il primo bersaglio del `makefile`, che definisce il bersaglio di default, è il programma di riduzione dei dati; esso dipende dal suo sorgente da tutti gli oggetti definiti dalla variabile `OBJ`, dal file di definizioni `commondef.f` e dalla routine C `readfile.o`; si noti il `.F` del sorgente, che significa che il file prima di essere compilato viene fatto passare attraverso il preprocessore C (cosa che non avviene per i `.f`) che permette di usare i comandi di compilazione condizionale del preprocessore C con la relativa sintassi. Sotto segue il comando di compilazione che sfrutta le variabili definite in precedenza per specificare quale compilatore e opzioni usare e specifica di nuovo gli oggetti e le librerie.

Il secondo bersaglio definisce le regole per la compilazione della routine in C; essa dipende solo dal suo sorgente. Si noti che per la compilazione vengono usate le variabili relative al

compilatore C. Si noti anche che se questa regola viene usata, allora lo sarà anche la precedente, dato che `riduzione` dipende da `readfile.o`.

Il terzo bersaglio è apparentemente incomprensibile dato che vi compare solo il riferimento alla variabile `OBJ` con una sola dipendenza e nessuna regola, essa però mostra le possibilità (oltre che la complessità) di `make` connesse alla presenza di quelle regole implicite a cui avevamo accennato.

Anzitutto una peculiarità di `make` è che si possono anche usare più bersagli per una stessa regola (nell'esempio quelli contenuti nella variabile `OBJ` che viene espansa in una lista); in questo caso la regola di costruzione sarà applicata a ciascuno che si potrà citare nella regola stessa facendo riferimento con la variabile automatica: `$$`. L'esempio usato per la nostra costruzione però sembra non avere neanche la regola di costruzione.

Questa mancanza sia di regola che di dipendenze (ad esempio dai vari sorgenti) illustra le capacità di funzionamento automatico di `make`. Infatti è facile immaginarsi che un oggetto dipenda da un sorgente, e che per ottenere l'oggetto si debba compilare quest'ultimo.

Il comando `make` sa tutto questo per cui quando un bersaglio è un oggetto (cioè ha un nome tipo `qualcosa.o`) non è necessario specificare il sorgente, ma il programma lo va a cercare nella directory corrente (ma è possibile pure dirgli di cercarlo altrove, il caso è trattato nel manuale). Nel caso specifico allora si è messo come dipendenza solo il file delle definizioni che viene incluso in ogni subroutine.

Inoltre come dicevamo in genere per costruire un oggetto si deve compilarne il sorgente; `make` sa anche questo e sulla base dell'estensione del sorgente trovato (che nel caso sarà un `qualcosa.f`) applica la regola implicita. In questo caso la regola è quella di chiamare il compilatore fortran applicato al file oggetto e al relativo sorgente, questo viene fatto usando la variabile `FC` che è una delle variabili standard usata dalle regole implicite (come `CC` nel caso di file `.c`); per una maggiore flessibilità poi la regola standard usa anche la variabile `FFLAGS` per specificare, a scelta dell'utente che non ha che da definirla, quali flag di compilazione usare (nella documentazione sono riportate tutte le regole implicite e le relative variabili usate).

In questo modo è stato possibile usare una sola riga per indicare la serie di dipendenze e relative compilazioni delle singole subroutine; inoltre con l'uso della variabile `OBJ` l'aggiunta di una nuova eventuale routine `nuova.f` comporta solo l'aggiunta di `nuova.o` alla definizione di `OBJ`.

D.2 Source Control Management

Uno dei problemi più comuni che si hanno nella programmazione è quella di poter disporre di un sistema che consenta di tenere conto del lavoro effettuato, di tracciare l'evoluzione del codice, e, soprattutto nel caso di progetti portati avanti da più persone, consentire un accesso opportunamente coordinato fra i vari partecipanti alla base comune dei sorgenti dello sviluppo.

I programmi che servono a questo scopo vanno sotto il nome comune di SCM (*Source Control Manager*), e ne esistono di diversi tipi con diverse filosofie progettuali, in particolare nelle modalità con cui gestiscono l'accesso alla base di codice comune da parte dei singoli programmatori che vi accedono.

Fra questi uno dei più usati, nonostante la sua architettura sia considerata superata, è Subversion, un sistema di archiviazione centralizzata del codice che consente di tenere traccia di tutte le modifiche e di condividere un archivio comune per progetti portati avanti da diverse persone.

D.2.1 Introduzione a Subversion

Subversion è basato sul concetto di *repository*, un archivio centralizzato in cui vengono riposti e da cui vengono presi i sorgenti dei programmi. L'archivio tiene traccia delle diverse versioni registrate; i programmatori inviano le modifiche usando una copia locale che hanno nella loro directory di lavoro.

Subversion può gestire più di un progetto all'interno di un singolo server, ciascuno dei quali viene associato ad un *repository* distinto, ma si possono anche creare sotto-progetti suddividendo un *repository* in diverse directory; ma ciascun progetto avrà meccanismi di controllo (ad esempio quelli che consentono di inviare email all'inserimento di nuovo codice) comuni.

Una delle caratteristiche che contraddistinguono Subversion dal suo predecessore CVS è quella di essere gestibile in maniera molto flessibile l'accesso al repository, che può avvenire sia in maniera diretta facendo riferimento alla directory in cui questo è stato installato che via rete, tramite diversi protocolli. L'accesso più comune è fatto direttamente via HTTP, utilizzando opportune estensioni del protocollo DAV, ma è possibile passare attraverso SSH o fornire un servizio di rete dedicato.²

In generale è comunque necessario preoccuparsi delle modalità di accesso al codice soltanto in fase di primo accesso al *repository*, che occorrerà identificare o con il pathname alla directory dove questo si trova o con una opportuna URL (con il comune accesso via web del tutto analoga a quella che si usa in un browser), dopo di che detto indirizzo sarà salvato nella propria copia locale dei dati ed il riferimento diventerà implicito.

Il programma prevede infatti che in ogni directory che si è ottenuta come copia locale sia presente una directory `.svn` contenente tutti i dati necessari al programma. Inoltre il programma usa la directory `.subversion` nella home dell'utente per mantenere le configurazioni generali del client e le eventuali informazioni di autenticazione.

Tutte le operazioni di lavoro sul *repository* vengono effettuate lato client tramite il comando `svn` che vedremo in sez. D.2.1 ma la creazione e la inizializzazione dello stesso (così come la gestione lato server) devono essere fatte tramite il comando `svnadmin` eseguito sulla macchina che lo ospita. In generale infatti il comando `svn` richiede che si faccia riferimento ad un *repository* (al limite anche vuoto) esistente e questo deve essere opportunamente creato.

Il comando `svnadmin` utilizza una sintassi che richiede sempre l'ulteriore specificazione di un sotto-comando, seguito da eventuali altri argomenti. L'inizializzazione di un *repository* (che sarà creato sempre vuoto) viene eseguita con il comando:

```
svnadmin create /path/to/repository
```

dove `/path/to/repository` è la directory dove verranno creati e mantenuti tutti i file, una volta creato il *repository* si potrà iniziare ad utilizzarlo ed inserirvi i contenuti con il comando `svn`.

Non essendo questo un testo di amministrazione di sistema non tratteremo qui i dettagli della configurazione del server per l'accesso via rete al *repository*, per i quali si rimanda alla documentazione del progetto ed alla documentazione sistemistica scritta per Truelite Srl.³

²esiste all'uopo il programma `svnserv`, ma il suo uso è sconsigliato per le scarse prestazioni e le difficoltà riscontrate a gestire accessi di utenti diversi; la modalità di accesso preferita resta quella tramite le estensioni al protocollo DAV.

³rispettivamente disponibili su svn.tigris.org e labs.truelite.it/truedoc.

D.2.2 Utilizzo di svn

Una volta che si abbia a disposizione un *repository* si potrà creare un nuovo progetto sottoposto a controllo di versione importando al suo interno i dati disponibili. In genere è pratica comune suddividere il contenuto di un repository in tre directory secondo il seguente schema:

trunk	contiene la versione corrente i sviluppo, su cui vengono effettuate normalmente le modifiche e gli aggiornamenti;
tags	contiene le diverse versioni <i>fotografate</i> ad un certo istante del processo di sviluppo, ad esempio in occasione del rilascio di una versione stabile, così che sia possibile identificarle facilmente;
branches	contiene <i>rami</i> alternativi di sviluppo, ad esempio quello delle correzioni eseguite ad una versione stabile, che vengono portati avanti in maniera indipendente dalla versione principale.

Questa suddivisione consente di sfruttare la capacità di Subversion di creare senza spesa copie diverse del proprio contenuto, pertanto in genere si pone il proprio progetto di sviluppo sotto **trunk**, e si copia quest'ultima in occasione delle varie versioni di rilascio in altrettante sottocartelle di **tags** e qualora si voglia aprire un ramo alternativo di sviluppo basterà copiarci il punto di partenza del ramo sotto **branches** e iniziare ad eseguire le modifiche su di esso.

Le operazioni di gestione di un progetto con Subversion vengono eseguite con il comando **svn**, che analogamente al precedente **svnadmin** utilizza una sintassi basata sulla specificazione degli opportuni sotto-comandi. Si sono riportati quelli più importanti in tab. D.1.

Sotto-comando	Significato	
import	–	Importa i file della directory corrente sul <i>repository</i> .
checkout	co	Scarica una versione del progetto dal <i>repository</i> .
commit	ci	Invia le modifiche effettuate localmente al <i>repository</i> .
add	–	Richiede l’aggiunta un file o una directory al <i>repository</i> .
remove	rm	Richiede la rimozione un file o una directory dal <i>repository</i> .
copy	cp	Richiede la copia un file o una cartella del progetto (mantenendone la storia).
move	mv	Richiede lo spostamento un file o una directory (equivalente ad un cp seguito da un rm).
update	–	Aggiorna la copia locale.
resolved	–	Rimuove una situazione di conflitto presente su un file.

Tabella D.1: Tabella riassuntiva dei principali sotto-comandi di svn.

In genere però è piuttosto raro iniziare un progetto totalmente da zero, è molto più comune avere una qualche versione iniziale dei propri file all'interno di una cartella. In questo caso il primo passo è quello di eseguire una inizializzazione del *repository* importando al suo interno quanto già esistente. Per far questo occorre eseguire il comando:

```
svn import [/pathname] URL
```

questo può essere eseguito direttamente nella directory contenente la versione iniziale dei propri sorgenti nel qual caso il comando richiede come ulteriore argomento la directory o la URL con la quale indicare il *repository* da usare. Alternativamente si può passare come primo argomento il pathname della directory da importare, seguito dall'indicazione della URL del *repository*.

Si tenga presente che l'operazione di importazione inserisce sul *repository* il contenuto completo della directory indicata, compresi eventuali file nascosti e sotto-directory. È anche

possibile eseguire l'importazione di più directory da inserire in diverse sezioni del *repository*, ma un tal caso ciascuna importazione sarà vista con una diversa *release*. Ad ogni operazione di modifica del *repository* viene infatti assegnato un numero progressivo che consente di identificarne la storia delle modifiche e riportarsi ad un dato punto della stessa in ogni momento successivo.⁴

Una volta eseguita l'importazione di una versione iniziale è d'uopo cancellare la directory originale e ripartire dal progetto appena creato. L'operazione di recuperare ex-novo di tutti i file che fanno parte di un progetto, chiamata usualmente *checkout*, viene eseguita con il comando:⁵

```
svn checkout URL [/pathname]
```

che creerà nella directory corrente una directory corrispondente al nome specificato in coda alla URL passata come argomento, scaricando l'ultima versione dei file archiviati sul repository; alternativamente si può specificare come ulteriore argomento la directory su cui scaricare i file.

Sia in caso di *import* che di *checkout* è sempre possibile operare su una qualunque sotto cartella contenuta all'interno di un *repository*, ignorando totalmente quello che sta al di sopra, basterà indicare in sede di importazione o di estrazione iniziale un pathname o una URL che identifichi quella parte del progetto.

Se quando si effettua lo scaricamento non si vuole usare la versione più aggiornata, ma una versione precedente si può usare l'opzione *-r* seguita da un numero che scaricherà esattamente quella *release*, alternativamente al posto del numero si può indicare una data, e verrà presa la *release* più prossima a quella data.

A differenza di CVS Subversion non supporta l'uso di *etichette* associate ad una certa versione del proprio progetto, per questo è invalso l'uso di strutturare il repository secondo lo schema illustrato inizialmente; è infatti molto semplice (e non comporta nessun tipo di aggravio) creare delle copie complete di una qualunque parte del *repository* su un'altra parte dello stesso, per cui se si è eseguito lo sviluppo sulla cartella *trunk* sarà possibile creare banalmente una versione con etichetta *label* (o quel che si preferisce) semplicemente con una copia eseguita con:

```
svn cp trunk tags/label
```

Il risultato di questo comando è la creazione della nuova cartella *label* sotto *tags*, che sarà assolutamente identica, nel contenuto (e nella sua storia) a quanto presente in *trunk* al momento dell'esecuzione del comando. In questo modo, una volta salvate le modifiche,⁶ si potrà ottenere la versione *label* del proprio progetto semplicemente eseguendo un *checkout* di *tags/label* in un'altra directory.⁷

Una volta creata la propria copia locale dei programmi, è possibile lavorare su di essi ponendosi nella relativa directory, e apportare tutte le modifiche che si vogliono ai file ivi presenti; due comandi permettono inoltre di schedulare la rimozione o l'aggiunta di file al *repository*:⁸

⁴a differenza di CVS Subversion non assegna un numero di versione progressivo distinto ad ogni file, ma un numero di *release* progressivo ad ogni cambiamento globale del repository, pertanto non esiste il concetto di versione di un singolo file, quanto di stato di tutto il *repository* ad un dato momento, è comunque possibile richiedere in maniera indipendente la versione di ogni singolo file a qualunque *release* si desideri.

⁵alternativamente si può usare l'abbreviazione *svn co*.

⁶la copia viene eseguita localmente verrà creata anche sul *repository* solo dopo un *commit*.

⁷ovviamente una volta presa la suddetta versione si deve aver cura di non eseguire nessuna modifica a partire dalla stessa, per questo se si deve modificare una versione etichettata si usa *branches*.

⁸a differenza di CVS si possono aggiungere e rimuovere, ed anche spostare con *svn mv*, sia file che directory.

```
svn add file1.c
svn remove file2.c
```

ma niente viene modificato sul *repository* fintanto che non viene eseguito il cosiddetto *commit* delle modifiche, vale a dire fintanto che non viene dato il comando:⁹

```
svn commit [file]
```

ed è possibile eseguire il *commit* delle modifiche per un singolo file, indicandolo come ulteriore argomento, mentre se non si indica nulla verranno inviate tutte le modifiche presenti.

Si tenga presente però che il *commit* non verrà eseguito se nel frattempo i file del *repository* sono stati modificati; in questo caso *svn* rileverà la presenza di differenze fra la propria *release* e quella del *repository* e chiederà che si effettui preventivamente un aggiornamento. Questa è una delle operazioni di base di Subversion, che in genere si compie tutte le volte che si inizia a lavorare, il comando che la esegue è:

```
svn update
```

Questo comando opera a partire dalla directory in cui viene eseguito e ne aggiorna il contenuto (compreso quello di eventuali sotto-directory) alla versione presente, scaricando le ultime versioni dei file esistenti o nuovi file o directory aggiunte, cancellando eventuali file e directory rimossi dal *repository*. Esso inoltre esso cerca, in caso di presenza di modifiche eseguite in maniera indipendente sulla propria copia locale, di eseguire un *raccordo* (il cosiddetto *merging*) delle stesse con quelle presenti sulla versione del *repository*.

Fintanto che sono state modificate parti indipendenti di un file di testo in genere il processo di *merging* ha successo e le modifiche vengono incorporate automaticamente in conseguenza dell'aggiornamento, ma quando le modifiche attengono alla stessa parte di un file nel ci si troverà di fronte ad un conflitto ed a quel punto sarà richiesto al “committente” di intervenire manualmente sui file per i quali sono stati rilevati i conflitti per risolverli.

Per aiutare il committente nel suo compito quando l'operazione di aggiornamento fallisce nel raccordo delle modifiche lascia sezioni di codice in conflitto opportunamente marcate e separate fra loro come nell'esempio seguente:

```
<<<<<<< .mine
      $(CC) $(CFLAGS) -o pamacq pamacq.c -lm
=====
      $(CC) $(CFLAGS) -o pamacq pamacq.c

>>>>>>> r.122
```

In questo caso si c'è stata una modifica sul file (mostrata nella parte superiore) incompatibile con quella fatta nel *repository* (mostrata nella parte inferiore). Prima di eseguire un *commit* occorrerà pertanto integrare le modifiche e salvare nuovamente il file rimuovendo i marcatori, inoltre prima che il *commit* ritorni possibile si dovrà esplicitare la risoluzione del conflitto con il comando:

```
svn resolved file
```

Infine per capire la situazione della propria copia locale si può utilizzare il comando *svn status* che confronta i file presenti nella directory locale rispetto alla ultima versione scaricata dal *repository* e per tutti quelli che non corrispondono stampa a schermo delle informazioni di stato nella forma di un carattere seguito dal nome del file, secondo quanto illustrato in tab. D.2.

⁹in genere anche questo viene abbreviato, con *svn ci*.

Flag	Significato
?	File sconosciuto.
M	File modificato localmente.
A	File aggiunto.
C	File con conflitto.

Tabella D.2: Caratteri associati ai vari stati dei file.

Appendice E

Ringraziamenti

Desidero ringraziare tutti coloro che a vario titolo e a più riprese mi hanno aiutato ed hanno contribuito a migliorare in molteplici aspetti la qualità di GaPiL. In ordine rigorosamente alfabetico desidero citare:

Alessio Frusciante per l'apprezzamento, le innumerevoli correzioni ed i suggerimenti per rendere più chiara l'esposizione.

Daniele Masini per la rilettura puntuale, le innumerevoli correzioni, i consigli sull'esposizione ed i contributi relativi alle calling convention dei linguaggi e al confronto delle diverse tecniche di gestione della memoria.

Mirko Maischberger per la rilettura, le numerose correzioni, la segnalazione dei passi poco chiari e soprattutto per il grande lavoro svolto per produrre una versione della guida in un HTML piacevole ed accurato.

Fabio Rossi per la rilettura, le innumerevoli correzioni, ed i vari consigli stilistici ed i suggerimenti per il miglioramento della comprensione di vari passaggi.

Infine, vorrei ringraziare il Firenze Linux User Group (FLUG), di cui mi pregio di fare parte, che ha messo a disposizione il repository CVS su cui era presente la prima versione della Guida, ed il relativo spazio web, e Truelite Srl, l'azienda che ho fondato e di cui sono amministratore, che fornisce le risorse per il nuovo repository Git ed per il sito del progetto: <https://gapil.gnulinix.it>.

Appendice F

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

[<http://fsf.org/>](http://fsf.org/)

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

F.1 Applicability and Definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “*Document*”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “*you*”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “*Modified Version*” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “*Secondary Section*” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “*Invariant Sections*” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “*Cover Texts*” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “*Transparent*” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “*Opaque*”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “*Title Page*” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “*publisher*” means any person or entity that distributes copies of the Document to the public.

A section “*Entitled XYZ*” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “*Acknowledgements*”, “*Dedications*”, “*Endorsements*”, or “*History*”.) To “*Preserve the Title*” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

F.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

F.3 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

F.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as

Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

F.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

F.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

F.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

F.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

F.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

F.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

F.11 Relicensing

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

F.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Indice analitico

Access Control List (ACL), 176–185
advertised window, 589, 795
algoritmo di Nagle, 720, 795
ANSI C, 13
append mode, 211
at-functions, 217–231
automount, 220

bind mount, 119–120
Block Started by Symbol (BSS), 33
broadcast, 581
broken-down time, 298–302
bucket filter, 735
buffer overflow, 251–252
buffer overrun, 43
busy poll, 701

calendar time, 291–292, 294–300
capabilities, 391–406
capabilities bounding set, 393–394
capability state, 400–405
child reaper, 414
clock tick, 292–293
close-on-exec, 80, 213
collegamento
 ciondolante, 130
 diretto, 127–129
 simbolico, 129–131
completely duplicate binding, 710
container, 418
context switch, 27
cooperative multitasking, 89
copy on write, 63
core dump, 311
costante
 ACL_GROUP_OBJ, 177
 ACL_GROUP, 177
 ACL_MASK, 177
 ACL_OTHER, 177
 ACL_TYPE_ACCESS, 181
 ACL_TYPE_DEFAULT, 181

ACL_USER_OBJ, 177
ACL_USER, 177
ADJ_ESTERROR, 298
ADJ_FREQUENCY, 298
ADJ_MAXERROR, 298
ADJ_OFFSET_SINGLESOT, 298
ADJ_OFFSET, 298
ADJ_STATUS, 298
ADJ_TICK, 298
ADJ_TIMECONST, 298
AF_ALG, 596
AF_APPLETALK, 596, 602
AF_ASH, 596
AF_ATMPVC, 596
AF_ATMSVC, 596
AF_AX25, 596
AF_BLUETOOTH, 596
AF_BRIDGE, 596
AF_CAIF, 596
AF_CAN, 596
AF_DECnet, 596
AF_ECONET, 596
AF_FILE, 596
AF_IB, 596
AF_IEEE802154, 596
AF_INET6, 596, 601
AF_INET, 596, 600
AF_IPX, 596
AF_IRDA, 596
AF_ISDN, 596
AF_IUCV, 596
AF_KCM, 596
AF_KEY, 596
AF_LLC, 596
AF_LOCAL, 596
AF_MAX, 596
AF_MPLS, 596
AF_NETBEUI, 596
AF_NETLINK, 596
AF_NETROM, 596

AF_NFC, 596
AF_PACKET, 596, 604
AF_PHONET, 596
AF_PPPOX, 596
AF_QIPCRTR, 596
AF_RDS, 596
AF_ROSE, 596
AF_ROUTE, 596
AF_RXRPC, 596
AF_SECURITY, 596
AF_SMC, 596
AF_SNA, 596
AF_TIPC, 596
AF_UNIX, 596, 602
AF_UNSPEC, 596
AF_VSOCK, 596
AF_WANPIPE, 596
AF_X25, 596
AIO_ALLDONE, 470
AIO_CANCELED, 470
AIO_NOTCANCELED, 470
AI_ADDRCONFIG, 684
AI_ALL, 684
AI_CANONNAME, 689
AI_DEFAULT, 684
AI_NUMERICHOST, 689
AI_NUMERICSERVICE, 689
AI_PASSIVE, 689
AI_V4MAPPED, 684
ARG_MAX, 267
ATADDR_BCAST, 603
ATEXIT_MAX, 29
ATPROTO_DDP, 602
AT_ANYNET, 602
AT_ANYNODE, 603
AT_EACCES, 220
AT_EMPTY_PATH, 220
AT_FDCWD, 219
AT_NO_AUTOMOUNT, 220
AT_REMOVEDIR, 220
AT_STATX_DONT_SYNC, 227
AT_STATX_FORCE_SYNC, 227
AT_STATX_SYNC_AS_STAT, 227
AT_SYMLINK_FOLLOW, 220
AT_SYMLINK_NOFOLLOW, 220
BOOT_TIME, 278
BRKINT, 379
BSDLY, 380
BUFSIZ, 260
BUS_ADRALN, 338
BUS_ADERR, 338
BUS_OBJERR, 338
CAP_AUDIT_CONTROL, 397
CAP_AUDIT_WRITE, 397
CAP_BLOCK_SUSPEND, 397
CAP_CHOWN, 397
CAP_CLEAR, 403
CAP_DAC_OVERRIDE, 397
CAP_DAC_READ_SEARCH, 397
CAP_EFFECTIVE, 402
CAP_FOWNER, 396–398
CAP_FSETID, 397
CAP_INHERITABLE, 402
CAP_IPC_LOCK, 397
CAP_IPC_OWNER, 397
CAP_KILL, 397
CAP_LEASE, 397
CAP_LINUX_IMMUTABLE, 397
CAP_MAC_ADMIN, 397
CAP_MAC_OVERRIDE, 397
CAP_MKNOD, 397
CAP_NET_ADMIN, 398
CAP_NET_BIND_SERVICE, 397
CAP_NET_BROADCAST, 397
CAP_NET_RAW, 397
CAP_PERMITTED, 402
CAP_SETFCAP, 397
CAP_SETGID, 397
CAP_SETPCAP, 396
CAP_SETUID, 397
CAP_SET, 403
CAP_SYSLOG, 397
CAP_SYS_ADMIN, 398
CAP_SYS_BOOT, 397
CAP_SYS_CHROOT, 397
CAP_SYS_MODULE, 397
CAP_SYS_NICE, 398
CAP_SYS_PACCT, 397
CAP_SYS_PTRACE, 397
CAP_SYS_RAWIO, 397
CAP_SYS_RESOURCE, 398
CAP_SYS_TIME, 397
CAP_SYS_TTY_CONFIG, 397
CAP_WAKE_ALARM, 397
CBAUDEX, 381
CBAUD, 381
CHAR_BIT, 266
CHAR_MAX, 266

CHAR_MIN, 266
CHILD_MAX, 267
CIBAUD, 381
CLD_CONTINUED, 338
CLD_DUMPED, 338
CLD_EXITED, 338
CLD_KILLED, 338
CLD_STOPPED, 338
CLD_TRAPPED, 338
CLOCAL, 381
CLOCKS_PER_SEC, 292
CLOCK_BOOTTIME_ALARM, 349
CLOCK_BOOTTIME, 349
CLOCK_MONOTONIC_RAW, 349
CLOCK_MONOTONIC, 349
CLOCK_PROCESS_CPUTIME_ID, 349
CLOCK_REALTIME_ALARM, 349
CLOCK_REALTIME, 349
CLOCK_THREAD_CPUTIME_ID, 349
CLONE_CHILD_CLEARTID, 416
CLONE_CHILD_SETTID, 416
CLONE_FILES, 416
CLONE_FS, 417
CLONE_IO, 417
CLONE_PARENT_SETTID, 417
CLONE_PARENT, 417
CLONE_PID, 417
CLONE_PTRACE, 417
CLONE_SETTLS, 417
CLONE_SIGHAND, 417
CLONE_STOPPED, 417
CLONE_SYSVSEM, 417
CLONE_THREAD, 417
CLONE_UNTRACED, 417
CLONE_VFORK, 417
CLONE_VM, 418
CMSPAR, 381
CPU_SETSIZE, 101
CRDLY, 380
CREAD, 381
CRTSCTS, 381
CSIZE, 381
CSTOPB, 381
C_ANY, 675
C_CHAOS, 675
C_CSNET, 675
C_HS, 675
C_IN, 675
DEAD_PROCESS, 278
DEFECHO, 382
DN_ACCESS, 459
DN_ATTRIB, 459
DN_CREATE, 459
DN_DELETE, 459
DN_MODIFY, 459
DN_MULTISHOT, 459
DN_RENAME, 459
DT_BLK, 139
DT_CHR, 139
DT_DIR, 139
DT_FIFO, 139
DT_LNK, 139
DT_REG, 139
DT_SOCK, 139
DT_UNKNOWN, 139
EAI_ADDRFAMILY, 689
EAI_AGAIN, 689
EAI_BADFLAGS, 689
EAI_FAIL, 689
EAI_FAMILY, 689
EAI_MEMORY, 689
EAI_NODATA, 689
EAI_NONAME, 689
EAI_SERVICE, 689
EAI_SOCKTYPE, 689
EAI_SYSTEM, 689
ECHOCTL, 382
ECHOE, 382
ECHOKE, 382
ECHOK, 382
ECHONL, 382
ECHOPRT, 382
ECHO, 382
EMPTY, 278
EPOLLERR, 443
EPOLLET, 443
EPOLLHUP, 443
EPOLLIN, 443
EPOLLONESHOT, 443
EPOLLOUT, 443
EPOLLPRI, 443
EPOLLRDHUP, 443
EPOLLWAKEUP, 443
EPOLL_CLOEXEC, 441
EPOLL_CTL_ADD, 442
EPOLL_CTL_DEL, 442
EPOLL_CTL_MOD, 442
ETH_P_ALL, 603

EXIT_FAILURE, 28
EXIT_SUCCESS, 28
FALLOC_FL_COLLAPSE_RANGE, 495
FALLOC_FL_INSERT, 495
FALLOC_FL_KEEP_SIZE, 495
FALLOC_FL_PUNCH_HOLE, 495
FALLOC_FL_ZERO_RANGE, 495
FASYNC, 203
FFDLY, 380
FIOASYNC, 239
FIOCLEX, 239
FIOGETOWN, 240
FIONBIO, 239
FIONCLEX, 239
FIONREAD, 240
FIOQSIZE, 240
FIOSETOWN, 240
FLUSHO, 382
FL_FLOCK, 422
FL_LEASE, 422
FL_POSIX, 422
FOPEN_MAX, 266
FPE_FLTDIV, 338
FPE_FLTINV, 338
FPE_FLOV, 338
FPE_FLTRES, 338
FPE_FLTSUB, 338
FPE_FLTUND, 338
FPE_INTDIV, 338
FPE_INTOV, 338
FSETLOCKING_BYCALLER, 262
FSETLOCKING_INTERNAL, 262
FSETLOCKING_QUERY, 262
F_ADD_SEALS, 237
F_DUPFD_CLOEXEC, 232
F_DUPFD, 232
F_GETFD, 232
F_GETFL, 233
F_GETLEASE, 236
F_GETLK, 233, 424
F_GETOWN_EX, 235
F_GETOWN, 234
F_GETPIPE_SZ, 237
F_GETSIG, 236
F_GET_FILE_RW_HINT, 237
F_GET_RW_HINT, 237
F_GET_SEALS, 237
F_LOCK, 429
F_NOTIFY, 236
F_OFD_GETLK, 233
F_OFD_SETLK, 234
F_OFD_SETLK, 233
F_OK, 166
F_OWNER_PGRP, 235
F_OWNER_PID, 235
F_OWNER_TID, 235
F_RDLCK, 424, 458
F_SETFD, 232
F_SETFL, 233
F_SETLEASE, 236
F_SETLK, 233, 424
F_SETLK, 233, 424
F_SETOWN_EX, 235
F_SETOWN, 234
F_SETPIPE_SZ, 237
F_SETSIG, 236
F_SET_FILE_RW_HINT, 238
F_SET_RW_HINT, 237
F_TEST, 430
F_TLOCK, 430
F_ULOCK, 430
F_UNLCK, 424, 458
F_WRLCK, 424, 458
GETALL, 529
GETNCNT, 529
GETPID, 529
GETVAL, 529
GETZCNT, 529
HARD_MAX, 555
HOST_NOT_FOUND, 677
HUPCL, 381
HZ, 292
ICANON, 382
ICRNL, 379
IEXTEN, 382
IFF_ALLMULTI, 728
IFF_AUTOMEDIA, 728
IFF_BROADCAST, 728
IFF_DEBUG, 728
IFF_DYNAMIC, 728
IFF_LOOPBACK, 728
IFF_MASTER, 728
IFF_MULTICAST, 728
IFF_NOARP, 728
IFF_NOTRAILERS, 728
IFF_POINTOPOINT, 728
IFF_PORTSEL, 728
IFF_PROMISC, 728

IFF_RUNNING, 728
IFF_SLAVE, 728
IFF_UP, 728
IFNAMSIZ, 701
IGNBRK, 379
IGNCR, 379
IGNPAR, 379
IIF_ALL, 191
IIF_BGRACE, 191
IIF_FLAGS, 191
IIF_IGRACE, 191
ILL_BADSTK, 338
ILL_COPROC, 338
ILL_ILLADR, 338
ILL_ILLOPC, 338
ILL_ILLOPN, 338
ILL_ILLTRP, 338
ILL_PRVOPC, 338
ILL_PRIVREG, 338
IMAXBEL, 379
IN6ADDR_ANY_INIT, 619
IN6ADDR_LOOPBACK_INIT, 619
INADDR_ANY, 619
INADDR_BROADCAST, 619
INADDR_LOOPBACK, 619
INADDR_NONE, 619
INET6_ADDRSTRLEN, 607
INET_ADDRSTRLEN, 607
INIT_PROCESS, 278
INLCR, 379
INPCK, 379
INT_MAX, 266
INT_MIN, 266
IN_ACCESS, 462
IN_ALL_EVENTS, 462
IN_ATTRIB, 462
IN_CLOSE_NOWRITE, 462
IN_CLOSE_WRITE, 462
IN_CLOSE, 462
IN_CREATE, 462
IN_DELETE_SELF, 462
IN_DELETE, 462
IN_DONT_FOLLOW, 462
IN_IGNORED, 464
IN_ISDIR, 464
IN_MASK_ADD, 462
IN_MODIFY, 462
IN_MOVED_FROM, 462
IN_MOVED_TO, 462
IN_MOVE_SELF, 462
IN_MOVE, 462
IN_ONESHOT, 462
IN_ONLYDIR, 462
IN_OPEN, 462
IN_Q_OVERFLOW, 464
IN_UNMOUNT, 464
IOPRIO_CLASS_BE, 105
IOPRIO_CLASS_IDLE, 104
IOPRIO_CLASS_RT, 105
IPCMNI, 514
IPC_CREAT, 516
IPC_EXCL, 517
IPC_INFO, 519
IPC_NOWAIT, 520
IPC_PRIVATE, 511
IPC_RMID, 518
IPC_SET, 519
IPC_STAT, 518
IPPORT_RESERVED, 616
IPPORT_USERRESERVED, 616
IPPROTO_IP, 713
IPPROTO_TCP, 719
IPPROTO_UDP, 725
IPRIO_WHO_PRGR, 104
IPRIO_WHO_PROCESS, 104
IPRIO_WHO_USER, 104
IPTOS_LOWDELAY, 778
IPTOS_MINCOST, 778
IPTOS_RELIABILITY, 778
IPTOS_THROUGHPUT, 778
IP_ADD_MEMBERSHIP, 714
IP_ADD_SOURCE_MEMBERSHIP, 714
IP_BLOCK_SOURCE, 714
IP_DROP_MEMBERSHIP, 715
IP_DROP_SOURCE_MEMBERSHIP, 715
IP_FREEBIND, 715
IP_HDRINCL, 715
IP_MINTTL, 716
IP_MSFILTER, 715
IP_MTU_DISCOVER, 716
IP_MTU, 716
IP_MULTICAST_IF, 717
IP_MULTICAST_LOOP, 717
IP_MULTICAST_TTL, 717
IP_OPTIONS, 717
IP_PKTINFO, 717
IP_PMTUDISC_DONT, 716
IP_PMTUDISC_DO, 716

IP_PMTUDISC_PROBE, 716
IP_PMTUDISC_WANT, 716
IP_RECVDSTADDR, 710
IP_RECVERR, 718
IP_RECVOPTS, 718
IP_RECVTOS, 718
IP_RECVTTL, 718
IP_RETOPTS, 718
IP_ROUTER_ALERT, 718
IP_TOS, 719
IP_TTL, 719
ISIG, 382
ISTRIP, 379
ITIMER_PROF, 325
ITIMER_REAL, 325
ITIMER_VIRTUAL, 325
IUCLC, 379
IUTF8, 379
IXANY, 379
IXOFF, 379
IXON, 379
KEXEC_ARCH_DEFAULT, 282
KEXEC_ON_CRASH, 282
KEXEC_PRESERVE_CONTEXT, 282
LINK_MAX, 269
LINUX_REBOOT_CMD_CAD_OFF, 280
LINUX_REBOOT_CMD_CAD_ON, 280
LINUX_REBOOT_CMD_HALT, 280
LINUX_REBOOT_CMD_KEXEC, 280
LINUX_REBOOT_CMD_POWER_OFF, 281
LINUX_REBOOT_CMD_RESTART2, 281
LINUX_REBOOT_CMD_RESTART, 281
LIO_NOP, 471
LIO_NOWAIT, 471
LIO_READ, 471
LIO_WAIT, 471
LIO_WRITE, 471
LLONG_MAX, 266
LLONG_MIN, 266
LOBLK, 381
LOCK_EX, 421
LOCK_NB, 421
LOCK_SH, 421
LOCK_UN, 421
LOGIN_PROCESS, 278
LOG_ALERT, 371
LOG_AUTHPRIV, 370
LOG_AUTH, 370
LOG_CONS, 371
LOG_CRIT, 371
LOG_CRON, 370
LOG_DAEMON, 370
LOG_DEBUG, 371
LOG_EMERG, 371
LOG_ERR, 371
LOG_FTP, 370
LOG_INFO, 371
LOG_KERN, 370
LOG_LOCAL0, 370
LOG_LOCAL7, 370
LOG_LPR, 370
LOG_MAIL, 370
LOG_NDELAY, 371
LOG_NEWS, 370
LOG_NOTICE, 371
LOG_NOWAIT, 371
LOG_ODELAY, 371
LOG_PERROR, 371
LOG_PID, 371
LOG_SYSLOG, 370
LOG_USER, 370
LOG_UUCP, 370
LOG_WARNING, 371
LONG_MAX, 266
LONG_MIN, 266
L_INCR, 207
L_SET, 207
L_XTND, 207
L_ctermid, 378
L_tmpnam, 148
MADV_DODUMP, 481
MADV_DOFORK, 481
MADV_DONTDUMP, 481
MADV_DONTFORK, 481
MADV_DONTNEED, 481
MADV_HUGEPAGE, 481
MADV_HWPOISON, 481
MADV_MERGEABLE, 481
MADV_NOHUGEPAGE, 481
MADV_NORMAL, 481
MADV_RANDOM, 481
MADV_REMOVE, 481
MADV_SEQUENTIAL, 481
MADV_SOFT_OFFLINE, 481
MADV_UNMERGEABLE, 481
MAP_32BIT, 474
MAP_ANONYMOUS, 474
MAP_ANON, 474

MAP_DENYWRITE, 474
MAP_EXECUTABLE, 474
MAP_FILE, 474
MAP_FIXED, 474
MAP_GROWSDOWN, 474
MAP_HUGETLB, 474
MAP_LOCKED, 474
MAP_NONBLOCK, 474
MAP_NORESERVE, 474
MAP_POPULATE, 474
MAP_PRIVATE, 474
MAP_SHARED, 474
MAP_UNINITIALIZED, 474
MAXSYMLINKS, 131
MAX_CANON, 269
MAX_INPUT, 269
MAX_SEC_IN_JIFFIES, 326
MB_LEN_MAX, 266
MCHECK_DISABLED, 44
MCHECK_FREE, 44
MCHECK_HEAD, 44
MCHECK_OK, 44
MCHECK_TAIL, 44
MCL_CURRENT, 41
MCL_FUTURE, 41
MCL_ONFAULT, 41
MINSIGSTKSZ, 357
MLOCK_ONFAULT, 41
MNT_DETACH, 125
MNT_EXPIRE, 125
MNT_FORCE, 125
MQ_PRIO_MAX, 558
MREMAP_MAYmove, 478
MSGMAX, 517
MSGMNB, 517
MSGMNI, 517
MSG_EXCEPT, 521
MSG_INFO, 519
MSG_NOERROR, 521
MSG_NOSIGNAL, 748
MSG_R, 513
MSG_STAT, 519
MSG_W, 513
MS_ASYNC, 477
MS_BIND, 119
MS_DIRSYNC, 121
MS_INVALIDATE, 477
MS_LAZYTIME, 122
MS_MANDLOCK, 122
MS_MGC_MSK, 118
MS_MGC_VAL, 118
MS_MOVE, 121
MS_NOATIME, 122
MS_NODEV, 122
MS_NODIRATIME, 122
MS_NOEXEC, 123
MS_NOSUID, 123
MS_PRIVATE, 121
MS_RDONLY, 123
MS_REC, 123
MS_RELATIME, 123
MS_REMOUNT, 119
MS_SHARED, 121
MS_SILENT, 124
MS_SLAVE, 121
MS_STRICTATIME, 124
MS_SYNCHRONOUS, 124
MS_SYNC, 477
MS_UNBINDABLE, 121
NAME_MAX, 269
NCCS, 382
NET_TCP_MAX_SYN_BACKLOG, 622
NEW_TIME, 278
NGROUPS_MAX, 267
NI_DGRAM, 693
NI_MAXHOST, 693
NI_MAXSERV, 693
NI_NAMEREQD, 693
NI_NOFQDN, 693
NI_NUMERICHOST, 693
NI_NUMERICSERV, 693
NLDLY, 380
NOFLSH, 382
NO_ADDRESS, 677
NO_DATA, 677
NO_RECOVERY, 677
OCRNL, 380
OFDEL, 380
OFILL, 380
OLCUC, 380
OLD_TIME, 278
ONLCR, 380
ONLRET, 380
ONOCR, 380
OPEN_MAX, 267
OPOST, 380
O_ACCMODE, 201
O_APPEND, 204

O_ASYNC, 204
O_CLOEXEC, 204
O_CREAT, 202
O_DIRECTORY, 202
O_DIRECT, 204
O_DSYNC, 204
O_EXCL, 202
O_EXEC, 201
O_EXLOCK, 202
O_LARGEFILE, 202
O_NDELAY, 204
O_NOATIME, 204
O_NOCTTY, 202
O_NOFOLLOW, 202
O_NONBLOCK, 204
O_PATH, 205
O_RDONLY, 201
O_RDWR, 201
O_READ, 201
O_SHLOCK, 202
O_SYNC, 204
O_TMPFILE, 203
O_TRUNC, 202
O_WRITE, 201
O_WRONLY, 201
PACKET_BROADCAST, 604
PACKET_HOST, 604
PACKET_MULTICAST, 604
PACKET_OTHERHOST, 604
PACKET_OUTGOING, 604
PAGECACHE_SIZE, 561
PAGE_SIZE, 31
PARENB, 381
PARMRK, 379
PARODD, 381
PATH_MAX, 269
PENDIN, 382
PID_MAX, 61
PIPE_BUF, 269
POLLERR, 438
POLLHUP, 438
POLLIN, 438
POLLMMSG, 438
POLLNVAL, 438
POLLOUT, 438
POLLPRI, 438
POLLRDBAND, 438
POLLRDHUP, 438
POLLRDNORM, 438
POLLWRBAND, 438
POLLWRNORM, 438
POLL_ERR, 338
POLL_HUP, 338
POLL_IN, 338
POLL_MSG, 338
POLL_OUT, 338
POLL_PRI, 338
POSIX_FADV_DONTNEED, 493
POSIX_FADV_NOREUSE, 493
POSIX_FADV_NORMAL, 493
POSIX_FADV_RANDOM, 493
POSIX_FADV_SEQUENTIAL, 493
POSIX_FADV_WILLNEED, 493
POSIX_MADV_DONTNEED, 482
POSIX_MADV_NORMAL, 482
POSIX_MADV_RANDOM, 482
POSIX_MADV_SEQUENTIAL, 482
POSIX_MADV_WILLNEED, 482
PRIO_MAX, 92
PRIO_MIN, 92
PRIO_PRGR, 93
PRIO_PROCESS, 93
PRIO_USER, 93
PROT_EXEC, 473
PROT_NONE, 473
PROT_READ, 473
PROT_WRITE, 473
PR_CAPBSET_DROP, 409
PR_CAPBSET_READ, 409
PR_ENDIAN_BIG, 410
PR_ENDIAN_LITTLE, 410
PR_ENDIAN_PPC_LITTLE, 410
PR_FPEMU_NOPRINT, 410
PR_FPEMU_SIGFPE, 410
PR_FP_EXC_ASYNC, 411
PR_FP_EXC_DISABLED, 410
PR_FP_EXC_DIV, 410
PR_FP_EXC_INV, 410
PR_FP_EXC_NONRECOV, 411
PR_FP_EXC_OVF, 410
PR_FP_EXC_PRECISE, 411
PR_FP_EXC_RES, 410
PR_FP_EXC_SW_ENABLE, 410
PR_FP_EXC_UND, 410
PR_GET_CHILD_SUBREAPER, 414
PR_GET_DUMPABLE, 410
PR_GET_ENDIAN, 410
PR_GET_FPEMU, 410

PR_GET_FPEXC, 411
PR_GET_KEEPCAPS, 411
PR_GET_NAME, 411
PR_GET_PDEATHSIG, 411
PR_GET_SECCOMP, 412
PR_GET_SECUREBITS, 412
PR_GET_TIMING, 413
PR_GET_TSC, 413
PR_MCE_KILL_CLEAR, 413
PR_MCE_KILL_DEFAULT, 414
PR_MCE_KILL_EARLY, 414
PR_MCE_KILL_GET, 414
PR_MCE_KILL_LATE, 414
PR_MCE_KILL_SET, 413
PR_SET_CHILD_SUBREAPER, 414
PR_SET_DUMPABLE, 409
PR_SET_ENDIAN, 410
PR_SET_FPEMU, 410
PR_SET_FPEXC, 410
PR_SET_KEEPCAPS, 411
PR_SET_NAME, 411
PR_SET_PDEATHSIG, 411
PR_SET_PTRACER_ANY, 412
PR_SET_PTRACER, 411
PR_SET_SECCOMP, 412
PR_SET_SECUREBITS, 412
PR_SET_TIMING, 412
PR_SET_TSC, 413
PR_SET_UNALIGN, 413
PR_TIMING_STATISTICAL, 412
PR_TIMING_TIMESTAMP, 412
PR_TSC_ENABLE, 413
PR_TSC_SIGSEGV, 413
PR_UNALIGN_NOPRINT, 413
PR_UNALIGN_SIGBUS, 413
PT_INTERP, 80
P_ALL, 75
P_PGID, 75
P_PID, 75
P_tmpdir, 148
QFMT_VFS_OLD, 191
QFMT_VFS_V0, 191
QFMT_VFS_V1, 191
QIF_ALL, 190
QIF_BLIMITS, 190
QIF_BTIME, 190
QIF_ILIMITS, 190
QIF_INODES, 190
QIF_ETIME, 190
QIF_LIMITS, 190
QIF_SPACE, 190
QIF_TIMES, 190
QIF_USAGE, 190
Q_GETFMT, 189
Q_GETINFO, 189
Q_GETQUOTA, 189
Q_GETSTATS, 189
Q_QUOTAOFF, 189
Q_QUOTAON, 189
Q_SETINFO, 189
Q_SETQUOTA, 189
Q_SYNC, 189
RENAME_EXCHANGE, 225
RENAME_NOREPLACE, 226
RENAME_WHITEOUT, 226
RES_AAONLY, 673
RES_BLAKE2, 673
RES_DEBUG, 673
RES_DEFAULT, 673
RES_DEFNAMES, 673
RES_DNSRCH, 673
RES_IGNTC, 673
RES_INIT, 673
RES_INSECURE1, 673
RES_INSECURE2, 673
RES_KEEPTSIG, 673
RES_NOALIASES, 673
RES_NOCHECKNAME, 673
RES_PRIMARY, 673
RES_RECURSE, 673
RES_ROTATE, 673
RES_STAYOPEN, 673
RES_USEVC, 673
RES_USE_INET6, 673
RLIMIT_AS, 285
RLIMIT_CORE, 286
RLIMIT_CPU, 286
RLIMIT_DATA, 286
RLIMIT_FSIZE, 286
RLIMIT_LOCKS, 286
RLIMIT_MEMLOCK, 286
RLIMIT_MSGQUEUE, 286
RLIMIT_NICE, 287
RLIMIT_NOFILE, 287
RLIMIT_NPROC, 287
RLIMIT_RSS, 287
RLIMIT_RTPRIO, 287
RLIMIT_RTIME, 287

RLIMIT_SIGPENDING, 287
RLIMIT_STACK, 288
RUN_LVL, 278
RUSAGE_CHILDREN, 284
RUSAGE_SELF, 284
RUSAGE_THREAD, 284
R_OK, 166
SA_NOCLDSTOP, 336
SA_NOCLDWAIT, 336
SA_NODEFER, 336
SA_NOMASK, 336
SA_ONESHOT, 336
SA_ONSTACK, 336
SA_RESETHAND, 336
SA_RESTART, 336
SA_SIGINFO, 336
SCHAR_MAX, 266
SCHAR_MIN, 266
SCHED_BATCH, 96
SCHED_FIFO, 96
SCHED_IDLE, 96
SCHED_OTHER, 96
SCHED_RR, 96
SECURE_KEEP_CAPS_LOCKED, 395
SECURE_KEEP_CAPS, 396
SECURE_NOROOT_LOCKED, 395
SECURE_NOROOT, 396
SECURE_NO_SETUID_FIXUP_LOCKED, 395
SECURE_NO_SETUID_FIXUP, 396
SEEK_CUR, 208
SEEK_DATA, 208
SEEK_END, 208
SEEK_HOLE, 208
SEEK_SET, 208
SEGV_ACCERR, 338
SEGV_MAPERR, 338
SEMAEM, 528
SEMMNI, 528
SEMMNS, 528
SEMMNU, 528
SEMMSL, 528
SEMOPM, 528
SEMUME, 528
SEMVMS, 528
SEM_FAILED, 564
SEM_INFO, 530
SEM_STAT, 530
SEM_UNDO, 531
SEM_VALUE_MAX, 564
SETALL, 529
SETVAL, 529
SFD_CLOEXEC, 447
SFD_NONBLOCK, 447
SHMALL, 538
SHMLBA, 538
SHMMAX, 538
SHMMIN, 538
SHMMNI, 538
SHMSEG, 538
SHM_HUGETLB, 536
SHM_INFO, 539
SHM_LOCK, 539
SHM_NORESERVE, 536
SHM_RDONLY, 540
SHM_REMAP, 540
SHM_RND, 540
SHM_STAT, 539
SHM_UNLOCK, 539
SHRT_MAX, 266
SHRT_MIN, 266
SHUT_RDWR, 659
SHUT_RD, 659
SHUT_WR, 659
SIGEV_NONE, 353
SIGEV_SIGNAL, 353
SIGEV_THREAD_ID, 353
SIGEV_THREAD, 353
SIGQUEUE_MAX, 346
SIGRTMAX, 345
SIGRTMIN, 345
SIGSTKSZ, 357
SIG_BLOCK, 341
SIG_DFL, 319
SIG_ERR, 321
SIG_IGN, 319
SIG_SETMASK, 341
SIG_UNBLOCK, 341
SIOCADDMULTI, 729
SIOCATMARK, 733
SIOCDELMULTI, 730
SIOCGIFCONF, 730
SIOCGIFFLAGS, 728
SIOCGIFHWADDR, 729
SIOCGIFINDEX, 728
SIOCGIFMAP, 729
SIOCGIFMETRIC, 729
SIOCGIFMTU, 729
SIOCGIFNAME, 728

SIOCGIFTXQLEN, 730
SIOCGPGRP, 726
SIOCGSTAMP, 726
SIOCINQ, 733
SIOCOUTQ, 733
SIOCSIFFLAGS, 729
SIOCSIFHWADDR, 729
SIOCSIFHWBROADCAST, 729
SIOCSIFMAP, 729
SIOCSIFMETRIC, 729
SIOCSIFMTU, 729
SIOCSIFNAME, 730
SIOCSIFTXQLEN, 730
SIOCSPPGRP, 726
SI_ASYNCIO, 338
SI_KERNEL, 338
SI_MESGQ, 338
SI_QUEUE, 338
SI_SIGIO, 338
SI_TIMER, 338
SI_TKILL, 338
SI_USER, 338
SOCK_CLOEXEC, 597
SOCK_DGRAM, 597
SOCK_NONBLOCK, 597, 624
SOCK_NOEXEC, 624
SOCK_PACKET, 597
SOCK_RAW, 597
SOCK_RDM, 597
SOCK_SEQPACKET, 597
SOCK_STREAM, 597
SOL_ICMPV6, 698
SOL_IPV6, 698
SOL_IP, 698
SOL_SOCKET, 698
SOL_TCP, 698
SOMAXCONN, 735
SO_ACCEPTCONN, 700
SO_ATTACH_FILTER, 701
SO_BINDTODEVICE, 701
SO_BROADCAST, 701
SO_BSDCOMPAT, 701
SO_BUSY_POLL, 701
SO_DEBUG, 702
SO_DETACH_FILTER, 702
SO_DOMAIN, 702
SO_DONTROUTE, 702
SO_ERROR, 702
SO_KEEPALIVE, 706–708
SO_LINGER, 711–712
SO_LOCK_FILTER, 703
SO_MARK, 703
SO_OOBINLINE, 703
SO_PASSCRED, 703
SO_PEEK_OFF, 703
SO_PEERCREC, 703
SO_PRIORITY, 704
SO_PROTOCOL, 704
SO_RCVBUFFORCE, 704
SO_RCVBUF, 704
SO_RCVLOWAT, 704
SO_RCVTIMEO, 705
SO_REUSEADDR, 708–711
SO_REUSEPORT, 711
SO_RXQ_OVFL, 705
SO_SNDBUFFORCE, 706
SO_SNDBUF, 705
SO_SNDLOWAT, 705
SO_SNDTIMEO, 706
SO_TIMESTAMP, 706
SO_TYPE, 706
SPLICE_F_GIFT, 487
SPLICE_F_MORE, 487
SPLICE_F_MOVE, 487
SPLICE_F_NONBLOCK, 487
SSIZE_MAX, 267
SS_DISABLE, 357
SS_ONSTACK, 357
STATX_ALL, 227
STATX_ETIME, 227
STATX_ATTR_APPEND, 229
STATX_ATTR_AUTOMOUNT, 229
STATX_ATTR_COMPRESSED, 229
STATX_ATTR_ENCRYPTED, 229
STATX_ATTR_IMMUTABLE, 229
STATX_ATTR_NODUMP, 229
STATX_BASIC_STATS, 227
STATX_BLOCKS, 227
STATX_BTIME, 227
STATX_CTIME, 227
STATX_GID, 227
STATX_INO, 227
STATX_MODE, 227
STATX_MTIME, 227
STATX_NLINK, 227
STATX_SIZE, 227
STATX_TYPE, 227
STATX_UID, 227

STATX__RESERVED, 227
STDERR_FILENO, 199
STDIN_FILENO, 199
STDOUT_FILENO, 199
STREAM_MAX, 267
SUID_DUMP_DISABLE, 410
SUID_DUMP_USER, 410
SYS_NMLN, 272
S_IFBLK, 153
S_IFCHR, 153
S_IFDIR, 153
S_IFIFO, 153
S_IFLNK, 153
S_IFMT, 153
S_IFREG, 153
S_IFSOCK, 153
S_IRGRP, 153
S_IROTH, 153
S_IRUSR, 153
S_IRWXG, 153
S_IRWXO, 153
S_IRWXU, 153
S_ISGID, 153
S_ISUID, 153
S_ISVTX, 153
S_IWGRP, 153
S_IWOTH, 153
S_IWUSR, 153
S_IXGRP, 153
S_IXOTH, 153
S_IXUSR, 153
TABDLY, 380
TCIFLUSH, 388
TCIOFF, 389
TCIOFLUSH, 388
TCION, 389
TCOFLUSH, 388
TCOOFF, 389
TCOON, 389
TCP_CONGESTION, 724
TCP_CORK, 720
TCP_DEFER_ACCEPT, 722
TCP_INFO, 723
TCP_KEEPCNT, 721
TCP_KEEPIIDLE, 721
TCP_KEEPIIDLE, 721
TCP_LINGER2, 722
TCP_MAXSEG, 720
TCP_MSS, 591
TCP_NODELAY, 719
TCP_QUICKACK, 724
TCP_SYNCNT, 722
TCP_WINDOW_CLAMP, 723
TCSADRAIN, 384
TCSAFLUSH, 384
TCSANOW, 384
TEXT_ABBREVIATE, 183
TEXT_ALL_EFFECTIVE, 183
TEXT_NUMERIC_IDS, 183
TEXT_SMART_INDENT, 183
TEXT_SOME_EFFECTIVE, 183
TFD_CLOEXEC, 454
TFD_NONBLOCK, 454
TFD_TIMER_ABSTIME, 454
TIMER_ABSTIME, 354
TIME_BAD, 297
TIME_DEL, 297
TIME_INS, 297
TIME_OK, 297
TIME_OOP, 297
TIME_WAIT, 297
TIOCGPRG, 365
TIOCSCTTY, 364
TIOCSGRP, 365
TMP_MAX, 148
TOSTOP, 382
TRAP_BRKPT, 338
TRAP_TRACE, 338
TRY_AGAIN, 677
TZNAME_MAX, 267
T_AAAA, 676
T_AFSDB, 676
T_ANY, 676
T_ATMA, 676
T_AXFR, 676
T_A, 676
T_CNAME, 676
T_EID, 676
T_GPOS, 676
T_HINFO, 676
T_ISDN, 676
T_IXFR, 676
T_KEY, 676
T_LOC, 676
T_MAILA, 676
T_MAILB, 676
T_MB, 676
T_MD, 676

T_MF, 676
T_MG, 676
T_MINFO, 676
T_MR, 676
T_MX, 676
T_NAPTR, 676
T_NIMLOC, 676
T_NSAP_PTR, 676
T_NSAP, 676
T_NS, 676
T_NULL, 676
T_NXT, 676
T_PTR, 676
T_PX, 676
T_RP, 676
T_RT, 676
T_SIG, 676
T_SOA, 676
T_SRV, 676
T_TSIG, 676
T_TXT, 676
T_WKS, 676
T_X25, 676
UCHAR_MAX, 266
UDP_CORK, 725
UDP_ENCAP, 725, 726
UINT_MAX, 266
ULLONG_MAX, 266
ULONG_MAX, 266
UMOUNT_NOFOLLOW, 125
USER_PROCESS, 278
USHRT_MAX, 266
UTIME_NOW, 160
UTIME_OMIT, 160
UTSLEN, 272
VDISCARD, 383
VDSUSP, 383
VEOF, 383
VEOL2, 383
VEOL, 383
VERASE, 383
VINTR, 383
VKILL, 383
VLNEXT, 383
VMIN, 383
VQUIT, 383
VREPRINT, 383
VSTART, 383
VSTATUS, 383

VSTOP, 383
VSUSP, 383
VSWTC, 383
VTDLY, 380
VTIME, 383
VWERASE, 383
WAIT_ANY, 73
WAIT_MYPGRP, 73
WCONTINUED, 73
WEXITED, 76
WNOHANG, 73, 76
WNOWAIT, 76
WSTOPPED, 76
WUNTRACED, 73
W_OK, 166
XATTR_CREATE, 175
XATTR_REPLACE, 175
XCASE, 382
X_OK, 166
_IOFBF, 259
_IOLBF, 259
_IONBF, 259
_PATH_UTMP, 277
_PATH_WTMP, 277
_SYS_NMLN, 272
_UTSNAME_DOMAIN_LENGTH, 272
_UTSNAME_LENGTH, 272
__WALL, 73
__WCLONE, 73
__WNOPTHREAD, 73
CPU affinity, 99–103

dangling link, 130
deadlock, 107
deep copy, 680
Denial of Service (DoS), 201, 202
directory di lavoro, 143–145
directory stream, 136–141
direttiva linguaggio C
 const, 49
 extern, 51
 register, 52
 static, 32
 union, 337
 volatile, 56
disciplina di linea, 387
Discretionary Access Control (DAC), 392
dnotify, 459–460
Domain Name Service (DNS), 669

dotted-decimal, 605

double free, 35

effetto ping-pong, 99

endianness, 56–58

epoll, 439–446

Executable and Linkable Format (ELF), 11

exit status, 28

Explicit Congestion Notification (ECN), 738

Extended Attributes, 171–176

file

hole, 208–209

di configurazione

/etc/fstab, 126

/etc/group, 273

/etc/gshadow, 273

/etc/inittab, 366

/etc/localtime, 300

/etc/mtab, 126, 127

/etc/passwd, 273

/etc/shadow, 273

/etc/timezone, 292

di dispositivo, 10–11, 145–147

di lock, 548–549

di sistema

/lib/ld-linux.so.1, 80

/lib/ld-linux.so.2, 80

/var/log/wtmp, 84

/var/run/utmp, 84

file di controllo (sotto /proc/sys)

fs/epoll/max_user_instances, 441

fs/epoll/max_user_watches, 441

fs/file-max, 398

fs/inotify/max_queued_events, 463

fs/inotify/max_user_instances, 460

fs/inotify/max_user_watches, 461

fs/lease-break-time, 458

fs/mqueue/msg_max, 555

fs/mqueue/msgsize_max, 555

fs/mqueue/queues_max, 555

fs/pipe-max-size, 237

fs/protected_hardlinks, 407

fs/protected_symlinks, 408

kernel/acct, 290

kernel/cap-bound, 393

kernel/domainname, 273

kernel/hostname, 273

kernel/msgmax, 517

kernel/msgmnb, 517

kernel/msgmni, 517

kernel/osrelease, 273

kernel/ostype, 273

kernel/pid_max, 61

kernel/printk, 373

kernel/rtsig-max, 346

kernel/sched_compat_yield, 99

kernel/sem, 514

kernel/shmall, 538

kernel/shmmax, 538

kernel/shmmni, 538

kernel/version, 273

net/core/dev_weight, 735

net/core/lo_cong, 735

net/core/message_burst, 734

net/core/message_cost, 734

net/core/mod_cong, 735

net/core/netdev_max_backlog, 735

net/core/no_cong_thresh, 735

net/core/no_cong, 735

net/core/optmem_max, 735

net/core/rmem_default, 734

net/core/rmem_max, 734

net/core/somaxconn, 735

net/core/wmem_default, 734

net/core/wmem_max, 734

net/ipv4/ip_always_defrag, 737

net/ipv4/ip_autoconfig, 736

net/ipv4/ip_default_ttl, 736

net/ipv4/ip_dynaddr, 736

net/ipv4/ip_forward, 736

net/ipv4/ip_local_port_range, 736

net/ipv4/ip_no_pmtu_disc, 736

net/ipv4/ip_nonlocal_bind, 737

net/ipv4/ipfrag_high_thresh, 737

net/ipv4/ipfrag_low_thresh, 737

net/ipv4/tcp_abort_on_overflow, 737

net/ipv4/tcp_adv_win_scale, 738

net/ipv4/tcp_app_win, 738

net/ipv4/tcp_dsack, 738

net/ipv4/tcp_ecn, 738

net/ipv4/tcp_fack, 738

net/ipv4/tcp_fin_timeout, 738

net/ipv4/tcp_frto, 738

net/ipv4/tcp_keepalive_intvl, 739

net/ipv4/tcp_keepalive_probes, 739

net/ipv4/tcp_keepalive_time, 739

net/ipv4/tcp_low_latency, 739

- net/ipv4/tcp_max_orphans, 739
- net/ipv4/tcp_max_syn_backlog, 739
- net/ipv4/tcp_max_tw_buckets, 739
- net/ipv4/tcp_mem, 740
- net/ipv4/tcp_orphan_retries, 740
- net/ipv4/tcp_reordering, 740
- net/ipv4/tcp_retrans_collapse, 740
- net/ipv4/tcp_retries1, 740
- net/ipv4/tcp_retries2, 740
- net/ipv4/tcp_rfc1337, 741
- net/ipv4/tcp_rmem, 741
- net/ipv4/tcp_sack, 741
- net/ipv4/tcp_stdurg, 741
- net/ipv4/tcp_syn_retries, 742
- net/ipv4/tcp_synack_retries, 741
- net/ipv4/tcp_syncookies, 742
- net/ipv4/tcp_timestamps, 742
- net/ipv4/tcp_tw_recycle, 742
- net/ipv4/tcp_tw_reuse, 742
- net/ipv4/tcp_window_scaling, 742
- net/ipv4/tcp_wmem, 742
- vm/memory_failure_early_kill, 413
- vm/, 216
- filesystem /proc
 - /proc/filesystems, 110
 - /proc/kmsg, 374
 - /proc/mounts, 127
 - /proc/self/cwd, 143
 - /proc/self/fs/quota/, 189
 - /proc/timer_list, 348
 - definizione, 271–273
- include
 - aio.h, 467
 - arpa/inet.h, 616
 - arpa/nameser.h, 675
 - arpa/nameser_compat.h, 675
 - assert.h, 26
 - ctype.h, 26
 - dirent.h, 26
 - errno.h, 26
 - fcntl.h, 26
 - float.h, 266
 - limits.h, 26
 - malloc.h, 26
 - netatalk/at.h, 602
 - netdb.h, 687
 - netinet/in.h, 600
 - netinet/ip.h, 713
 - netinet/tcp.h, 719
 - netinet/udp.h, 725
 - paths.h, 277
 - pwd.h, 274
 - resolv.h, 672
 - sched.h, 98
 - semaphore.h, 566
 - setjmp.h, 26
 - signal.h, 26
 - socket.h, 595
 - stdarg.h, 26
 - stdio.h, 26
 - stdlib.h, 26
 - string.h, 26
 - sys/capability.h, 399
 - sys/ioctl.h, 239
 - sys/ipc.h, 512
 - sys/msg.h, 517
 - sys/resource.h, 282
 - sys/select.h, 435
 - sys/stat.h, 151
 - sys/syscall.h, 28
 - sys/sysmacros.h, 146
 - sys/time.h, 93
 - sys/timex.h, 297
 - sys/types.h, 13
 - sys/un.h, 602
 - time.h, 26
 - times.h, 26
 - unistd.h, 26
 - utmp.h, 26
 - utmpx.h, 279
 - speciali, 10–11, 145–147
 - file attributes, 163
 - file capabilities, 392–396
 - file descriptor, 11–12, 197–199
 - file descriptor flags, 213
 - file descriptor set, 433–434
 - file descriptor table, 198
 - file lease, 457–459
 - file locking, 419–431
 - file status flags, 201
 - file stream, 12, 240–241
 - file table, 197–199
 - Fully Qualified Domain Name (FQDN), 674
 - funzione
 - _Exit, 29
 - __fbufsize, 260
 - __flbf, 260
 - __freadable, 258

__freading, 259
__fsetlocking, 262
__fwritable, 258
__fwriting, 259
_exit, 29
_flushlbf, 261
_sysctl, 270
abort, 324
accept4, 624
accept, 623
access, 165
acct, 290
acl_check, 185
acl_copy_entry, 185
acl_copy_ext, 184
acl_copy_int, 184
acl_create_entry, 185
acl_delete_def_file, 185
acl_delete_entry, 185
acl_dup, 180
acl_free, 180
acl_from_mode, 180
acl_from_text, 181
acl_get_entry, 185
acl_get_fd, 181
acl_get_file, 181
acl_get_permset, 185
acl_get_qualifier, 185
acl_get_tag_type, 185
acl_init, 179
acl_set_file, 184
acl_set_permset, 185
acl_set_qualifier, 185
acl_set_tag_type, 185
acl_size, 184
acl_to_any_text, 183
acl_to_text, 182
acl_valid, 185
addmntent, 126
adjtimex, 296
adjtime, 296
aio_cancel, 470
aio_error, 469
aio_fsync, 469
aio_read, 468
aio_return, 469
aio_suspend, 470
aio_write, 468
alarm, 324
aligned_alloc, 43
alloca, 36
alphasort, 141
asctime, 299
asprintf, 256
atexit, 29
bdflush, 216
bind, 618
brk, 37
calloc, 34
cap_clear_flag, 401
cap_clear, 401
cap_compare, 402
cap_dup, 401
cap_free, 401
cap_from_name, 404
cap_from_text, 404
cap_get_flag, 402
cap_get_pid, 405
cap_get_proc, 405
cap_init, 400
cap_set_flag, 402
cap_set_proc, 405
cap_to_name, 404
cap_to_text, 403
capgetp, 406
capget, 398
capsetp, 406
capset, 398
cfgetispeed, 387
cfgetospeed, 387
cfmakeraw, 387
cfree, 35
cfsetispeed, 386
cfsetospeed, 386
cfsetspeed, 386
chdir, 144
chmod, 166
chown, 169
chroot, 193
clearenv, 50
clearerr_unlocked, 246
clearerr, 246
clock_getcpuclockid, 350
clock_getres, 350
clock_gettime, 348
clock_nanosleep, 355
clock_settime, 348
clock, 293

clone, 415
closedir, 140
closelog, 372
close, 206
connect, 619
creat, 206
ctermid, 377
ctime, 299
daemon, 368
dirfd, 136
dprintf, 257
drand48, 15
dup2, 215
dup3, 215
dup, 214
eaccess, 166
endfsent, 126
endgrent, 276
endmntent, 126
endnetent, 686
endprotoent, 686
endpwent, 276
endservent, 685
endutent, 277
endutxent, 279
epoll_create1, 440
epoll_create, 440
epoll_ctl, 441
epoll_pwait, 445
epoll_wait, 444
error_at_line, 306
error, 305
euidaccess, 166
execle, 78
execlp, 78
execl, 78
execveat, 230
execve, 77
execvp, 78
execv, 78
exit, 28
faccessat, 221
fallocate, 495
fchdir, 145
fchmodat, 220
fchmod, 166
fchownat, 220
fchown, 169
fcloseall, 245
fclose, 245
fcntl, 232
fdatasync, 216
fdopendir, 137
fdopen, 243
feof, 246
ferror, 246
fexecve, 230
fflush, 260
fgetc, 249
fgetgrent_r, 276
fgetgrent, 276
fgetpos, 247
fgetpwent_r, 276
fgetpwent, 276
fgets, 251
fgetwc, 250
fgetws, 252
fgetxattr, 174
fileno, 258
flistxattr, 175
flockfile, 261
flock, 420
fmtmsg, 15
fopen, 243
fork, 62
fpathconf, 270
fprintf, 254
fpurge, 261
fputc, 250
fputs, 252
fputws, 252
fread, 248
freeaddrinfo, 692
freehostent, 683
free, 35
fremovexattr, 176
freopen, 243
fscanf, 257
fseek, 247
fsetpos, 247
fsetxattr, 174
fstatat, 222
fstatfs, 126
fstat, 150
fsync, 216
ftell, 247
ftok, 512
ftruncate, 154, 155

ftrylockfile, 262
funlockfile, 261
futimens, 159
futimesat, 219, 222
futimes, 158
fwrite, 248
gai_strerror, 689
get_avphys_pages, 289
get_current_dir_name, 144
get_phys_pages, 289
getaddrinfo, 687
getc_unlocked, 263
getchar_unlocked, 263
getchar, 249
getcwd, 143
getc, 249
getdelim, 254
getegid, 82
getenv, 48
geteuid, 82
getfsent, 126
getfsfile, 126
getfsspec, 126
getgid, 82
getgrent_r, 276
getgrent, 276
getgrgid, 275
getgrnam, 275
getgrouplist, 88
getgroups, 88
gethostbyaddr_r, 682
gethostbyaddr, 682
gethostbyname2_r, 681
gethostbyname2, 678
gethostbyname_r, 681
gethostbyname, 677
gethostent_r, 683
gethostent, 682
getipnodebyaddr, 683
getipnodebyname, 683
getitimer, 327
getline, 253
getloadavg, 290
getmntent_r, 126
getmntent, 126
getnameinfo, 692
getnetbyaddr, 684
getnetbyname, 684
getnetent, 686
getopt, 45
getpagesize, 289
getpeername, 625
getpgid, 361
getpgrp, 361
getpid, 61
getppid, 61
getpriority, 92
getprotobyaddr, 684
getprotobyname, 684
getprotoent, 686
getpwent_r, 276
getpwent, 276
getpwnam, 274
getpwuid, 274
getresgid, 86
getresuid, 86
getrlimit, 285
getrusage, 283
getservbyname, 684
getservbyport, 684
getservent, 685
getsid, 361
getsockname, 625
getsockopt, 699
gets, 251
gettimeofday, 295
getuid, 82
getutent, 277
getutid, 277
getutline, 277
getutxent, 279
getutxid, 279
getutxline, 279
getwchar, 250
getwc, 250
getwd, 144
getw, 250
getxattr, 174
gmtime, 300
hasmntopt, 126
herror, 677
hsearch, 15
hstrerror, 677
htonl, 605
htons, 605
inet_addr, 606
inet_aton, 606
inet_lnaof, 607

inet_makeaddr, 607
inet_netof, 607
inet_ntoa, 606
inet_ntop, 607
inet_pton, 607
initgroups, 89
inotify_add_watch, 461
inotify_init, 460
inotify_rm_watch, 462
ioctl, 238
ioprio_get, 103
ioprio_set, 103
isatty, 377
kexec_load, 281
killpg, 324
kill, 323
klogctl, 373
lchown, 169
lgetxattr, 174
linkat, 223
link, 127
lio_listio, 471
listen, 621
listxattr, 175
llistxattr, 175
localtime, 300
lockf, 429
logwtmp, 279
longjmp, 55
lremovexattr, 176
lseek64, 21
lseek, 207
lsetxattr, 174
lstat, 150
lutimes, 158
madvise, 480
malloc, 34
mcheck, 43
memalign, 42
memcpy, 22
memmove, 22
mempcpy, 22
memset, 22
mincore, 38
mkdirat, 219
mkdir, 134
mkdtemp, 150
mkfifoat, 219
mkfifo, 147
mknodat, 219
mknod, 145
mkostemps, 150
mkostemp, 149, 150
mkstemps, 150
mkstemp, 149, 150
mktemp, 149
mktime, 300
mlock2, 41
mlockall, 40
mlock, 40
mmap, 472
mount, 117
mprobe, 44
mprotect, 477
mq_close, 556
mq_getattr, 556
mq_notify, 559
mq_open, 553
mq_receive, 558
mq_send, 557
mq_setattr, 556
mq_timedreceive, 558
mq_timedsend, 557
mq_unlink, 556
mremap, 478
msgctl, 518
msgget, 516
msgrcv, 520
msgsnd, 519
msync, 476
munlockall, 40
munlock, 40
munmap, 477
nanosleep, 328
nice, 92
ntohl, 605
ntohs, 605
on_exit, 30
openat, 218
opendir, 136
openlog, 370
open, 199
pathconf, 270
pause, 327
pclose, 502
perror, 304
pipe2, 498
pipe, 497

poll, 437
popen, 502
posix_fadvise, 492
posix_fallocate, 494
posix_madvise, 480
posix_memalign, 43
posix_trace_event, 343
ppoll, 439
prctl, 409
preadv, 483
pread, 210
printf, 254
prlimit, 288
pselect, 435
psignal, 318
pthread_getcpuclockid, 351
putc_unlocked, 263
putchar_unlocked, 263
putchar, 250
putc, 250
putenv, 49
putgrent, 276
putpwent, 276
puts, 252
pututline, 277
pututxline, 279
putw, 250
pvalloc, 42
pwritev, 483
pwrite, 211, 494
qsort, 141
quotactl, 188
raise, 322
readahead, 492
readdir_r, 138
readdir, 137
readlinkat, 219
readlink, 131
readv, 482
read, 209
realloc, 35
reboot, 280
recvfrom, 748
remap_file_pages, 479
removexattr, 176
remove, 133
renameat2, 225
renameat, 225
rename, 133
res_init, 672
res_ninit, 673
res_nquery, 674
res_nsearch, 674
res_query, 674
res_search, 674
rewinddir, 140
rewind, 247
rmdir, 135
sbrk, 38
scandirat, 219
scandir, 140
scanf, 257
sched_get_priority_max, 96
sched_get_priority_min, 96
sched_getaffinity, 102
sched_getparam, 97
sched_getscheduler, 98
sched_rr_get_interval, 98
sched_setaffinity, 100
sched_setparam, 97
sched_setscheduler, 95
sched_yield, 98
seekdir, 139
select, 433
sem_close, 567
sem_destroy, 568
sem_getvalue, 566
sem_init, 568
sem_open, 564
sem_post, 566
sem_timedwait, 566
sem_trywait, 565
sem_unlink, 567
sem_wait, 565
semctl, 527
semget, 525
semop, 530
semtimedop, 531
sendfile, 484
sendto, 747
setbuffer, 260
setbuf, 260
setegid, 86
setenv, 49
seteuid, 86
setfsent, 126
setfsgid, 87
setfsuid, 87

setgid, 83
setgrent, 276
setgroups, 88
sethostent, 681
setitimer, 325
setjmp, 55
setlinebuf, 260
setlogmask, 372
setmntent, 126
setnetent, 686
setpgid, 362
setpgrp, 362
setpriority, 93
setprotoent, 686
setpwent, 276
setregid, 85
setresgid, 86
setresuid, 86
setreuid, 85
setrlimit, 285
setservent, 685
setsid, 363
setsockopt, 698
settimeofday, 295
setuid, 83
setutent, 277
setutxent, 279
setvbuf, 259
setxattr, 174
shm_open, 561
shm_unlink, 562
shmat, 539
shmctl, 538
shmdt, 541
shmget, 536
shutdown, 658
sigaction, 335
sigaddset, 334
sigaltstack, 357
sigandset, 334
sigdelset, 334
sigemptyset, 334
sigfillset, 334
sigisemptyset, 334
sigismember, 334
siglongjmp, 358
signalfd4, 447
signalfd, 447
signal, 321
sigorset, 334
sigpending, 356
sigprocmask, 340
sigqueue, 346
sigreturn, 407
sigsetjmp, 358
sigset, 343
sigsuspend, 341
sigtimedwait, 347
sigwaitinfo, 347
sigwait, 346
sleep, 327
snprintf, 254
socketpair, 510
socket, 594
splice, 486
sprintf, 254
sscanf, 257
statfs, 126
statx, 226
stat, 150
stime, 295
stpcpy, 22
strcat, 22
strcmp, 141
strcoll, 141
strcpy, 22
strerror_r, 304
strerror, 303
strftime, 301
strncat, 22
strncpy, 22
strptime, 301
strsignal, 318
strtol, 304
symlinkat, 219
symlink, 129
syncfs, 217
sync, 216
sys_clone, 415
syscall, 27
sysconf, 268
sysctl, 270
syslog, 370
sysv_signal, 322
tcdrain, 388
tcflow, 389
tcflush, 388
tcgetattr, 383

- tcgetpgrp, 364
- tcsendbreak, 388
- tcsetattr, 383
- tcsetpgrp, 364
- tee, 490
- telldir, 140
- tempnam, 148
- timer_create, 351
- timer_delete, 355
- timer_getoverrun, 354
- timer_gettime, 355
- timer_settime, 353
- timerfd_create, 453
- timerfd_gettime, 454
- timerfd_settime, 454
- times, 293
- time, 295
- tmpfile, 148
- tmpnam_r, 148
- tmpnam, 147
- truncate, 154
- ttynam_r, 377
- ttynam, 377
- tzset, 300
- umask, 168
- umount2, 125
- umount, 124
- uname, 272
- ungetc, 251
- unlinkat, 221
- unlink, 132
- unsetenv, 50
- updwtmpx, 279
- updwtmp, 279
- usleep, 328
- utimes, 158
- utime, 157
- utmpname, 277
- utmpxname, 279
- valloc, 42
- vasprintf, 256
- vdprintf, 257
- versionsort, 141
- vfork, 68
- vfprintf, 256
- vfscanf, 257
- vmsplice, 489
- vprintf, 256
- vscanf, 257
- vsnprintf, 256
- vsprintf, 256
- vsscanf, 257
- vsyslog, 371
- wait3, 76
- wait4, 76
- waitid, 74
- waitpid, 72
- wait, 71
- writev, 482
- write, 211
- funzioni
 - signal safe*, 343
 - variadic*, 52–54
 - rientranti, 107–108
- FUSE*, 126
- Group ID (GID)*, 81
- half-close*, 658–659
- hard link*, 127–129
- header file*, 25–26
- heap*, 32–33, 37–38
- High Resolution Timer (HRT)*, 348
- huge page*, 31
- inode*, 109–115
- inotify*, 459–467
- insieme di processori, 100–102
- interface index*, 728
- istruzione linguaggio C
 - goto, 54
 - if, 56
 - return, 28
 - switch, 46, 56
 - while, 56
- jiffies*, 292
- link-loader*, 23–24
- linked list*, 517
- Linux Security Modules (LSM)*, 391
- macro
 - CAP_DIFFERS, 402
 - CPU_ALLOC_SIZE, 102
 - CPU_ALLOC, 102
 - CPU_AND, 102
 - CPU_CLR, 101
 - CPU_COUNT, 101

CPU_EQUAL, 102
CPU_FREE, 102
CPU_ISSET, 101
CPU_OR, 102
CPU_SET, 101
CPU_XOR, 102
CPU_ZERO, 101
DTTOIF, 139
FD_CLR, 433
FD_ISSET, 433
FD_SETSIZE, 433
FD_SET, 433
FD_ZERO, 433
IFTODT, 139
IN6ADDR_ANY_INIT, 601
IN6ADDR_LOOPBACK_INIT, 601
IOPRIO_PRIO_CLASS, 104
IOPRIO_PRIO_DATA, 104
IOPRIO_PRIO_VALUE, 104
LOG_MASK, 372
LOG_UPTO, 372
NSIG, 313
QCMD, 188
SOCK_DEBUGGING, 702
STATE_TRACE, 702
S_ISBLK, 152
S_ISCHR, 152
S_ISDIR, 152
S_ISFIFO, 152
S_ISLNK, 152
S_ISREG, 152
S_ISSOCK, 152
WCOREDUMP, 75
WEXITSTATUS, 75
WIFCONTINUED, 75
WIFEXITED, 75
WIFSIGNALED, 75
WIFSTOPPED, 75
WSTOPSIG, 75
WTERMSIG, 75
_ATFILE_SOURCE, 22
_BSD_SOURCE, 19
_DEFAULT_SOURCE, 19
_DIRENT_HAVE_D_NAMLEN, 138
_DIRENT_HAVE_D_OFF, 138
_DIRENT_HAVE_D_RECLEN, 138
_DIRENT_HAVE_D_TYPE, 138
_FILE_OFFSET_BITS, 21
_FORTIFY_SOURCE, 22
_GNU_SOURCE, 20
_ISOC11_SOURCE, 20
_ISOC99_SOURCE, 20
_LARGEFILE64_SOURCE, 21
_LARGEFILE_SOURCE, 21
_POSIX_ARG_MAX, 267
_POSIX_ASYNCHRONOUS_IO, 467
_POSIX_CHILD_MAX, 267
_POSIX_CPUTIME, 348
_POSIX_C_SOURCE, 18
_POSIX_JOB_CONTROL, 268
_POSIX_LINK_MAX, 270
_POSIX_MAX_CANON, 270
_POSIX_MAX_INPUT, 270
_POSIX_MEMLOCK_RANGE, 40
_POSIX_MONOTONIC_CLOCK, 348
_POSIX_NAME_MAX, 270
_POSIX_NGROUPS_MAX, 267
_POSIX_OPEN_MAX, 267
_POSIX_PATH_MAX, 270
_POSIX_PIPE_BUF, 270
_POSIX_PRIORITIZED_IO, 468
_POSIX_PRIORITY_SCHEDULING, 468
_POSIX_RTSIG_MAX, 267
_POSIX_SAVED_IDS, 268
_POSIX_SIGQUEUE_MAX, 346
_POSIX_SOURCE, 18
_POSIX_SSIZE_MAX, 267
_POSIX_STREAM_MAX, 267
_POSIX_THREAD_CPUTIME, 348
_POSIX_THREAD_SAFE_FUNCTIONS, 261
_POSIX_TIMERS, 348
_POSIX_TZNAME_MAX, 267
_POSIX_VERSION, 268
_REENTRANT, 22
_SEM_SEMUN_UNDEFINED, 529
_SVID_SOURCE, 19
_THREAD_SAFE, 22
_XOPEN_SOURCE_EXTENDED, 20
_XOPEN_SOURCE, 20
__STRICT_ANSI__, 18
__va_copy, 54
gnu_dev_major, 146
gnu_dev_makedev, 146
gnu_dev_minor, 146
major, 147
makedev, 147
minor, 147
offsetof, 139

- va_arg, 52–53
- va_copy, 54
- va_end, 53
- va_start, 52–53
- magic number, 11
- major number, 146–147
- Mandatory Access Control (DAC), 391
- mandatory locking, 430–431
- maschera dei segnali, 340–343
- Maximum Segment Lifetime (MSL), 613
- Maximum Segment Size (MSS), 795
- Maximum Transfer Unit (MTU), 590–591
- memoria virtuale, 31–32, 38–42
- memory leak, 36
- memory locking, 39–42
- memory mapping, 472–480, 551–552
- minor number, 146–147
- modo canonico, 376
- modo non canonico, 376
- modo promiscuo, 728
- mount peer group, 120
- mount point, 117–118
- multicast, 581, 785–786
- namespace, 418
- Name Service Switch (NSS), 670–671
- netfilter, 703
- Network Address Translation (NAT), 742
- numeri di sequenza, 610
- out-of-band, 761–762
- overlay filesystem, 226
- page fault, 32
- page table, 31
- paginazione, 31–32
- Parent Process ID (PPID), 61–62
- pathname, 9–10, 110–111
 - assoluto, 9, 10
 - relativo, 9, 10
- pathname resolution, 9–10, 110–111
- Path MTU, 590–591, 716–717
- polling, 432
- POSIX IPC names, 552
- POSIX Timer API, 348–356
- preemptive multitasking, 4, 89
- prefaulting, 479
- process group, 361–363
- process group leader, 362
- process group orphaned, 365
- Process ID (PID), 61–62
- process table, 60–61, 197–198
- process time, 291–292
- protected hardlinks, 407
- protected symlinks, 408
- race condition, 106–107
- read-ahead, 492
- Request For Comment (RFC), 584
- Resident Set Size (RSS), 287
- resolver, 669–677
- Round Trip Time (RTT), 660
- salto non-locale, 54–56
- scheduler, 4, 61, 89–103
- securebits, 395–396
- secure computing mode, 407
- segmento
 - dati, 32–33, 37–38
 - testo, 32–33
- segment violation, 32
- segnale
 - SIGABRT, 314
 - SIGALRM, 315
 - SIGBUS, 314
 - SIGCHLD, 316
 - SIGCLD, 316
 - SIGCONT, 316
 - SIGEMT, 314
 - SIGFPE, 313
 - SIGHUP, 315
 - SIGILL, 313
 - SIGINFO, 318
 - SIGINT, 314
 - SIGIOT, 312
 - SIGIO, 316
 - SIGKILL, 315
 - SIGLOST, 317
 - SIGPIPE, 317
 - SIGPOLL, 316
 - SIGPROF, 315
 - SIGPWR, 318
 - SIGQUIT, 314
 - SIGSEGV, 314
 - SIGSTKFLT, 318
 - SIGSTOP, 316
 - SIGSYS, 314
 - SIGTERM, 314
 - SIGTRAP, 314

- SIGTSTP, 316
- SIGTTIN, 316
- SIGTTOU, 317
- SIGUNUSED, 312
- SIGURG, 316
- SIGUSR1, 317
- SIGUSR2, 317
- SIGWINCH, 318
- SIGXCPU, 317
- SIGXFSZ, 317
- SIVGTALRM, 315
- self-pipe trick*, 436
- sezione critica, 107
- sgid bit*, 164
- shared objects*, 23
- shared subtree*, 120–121
- side effects*, 101
- signal driven I/O*, 455–457
- signal set*, 333–335
- silly window syndrome*, 720
- socket
 - definizione, 593–594
 - locali, 760
- socket pair*, 616
- sparse file*, 208–209
- stack*, 32–33
- standard error*, 198–199
- standard input*, 198–199
- standard output*, 198–199
- sticky bit*, 164–165
- struttura dati
 - addrinfo, 687
 - aiocb, 467
 - at_addr, 603
 - cap_user_data_t, 399
 - cap_user_header_t, 399
 - dirent, 138
 - dqblk, 190
 - dqinfo, 191
 - epoll_event, 443
 - f_owner_ex, 235
 - flock, 423
 - fstab, 126
 - group, 275
 - hostent, 678
 - ifconf, 730
 - ifmap, 730
 - ifreq, 727
 - in6_addr, 601
 - in_addr, 600
 - inotify_event, 463
 - iovec, 483
 - ip_mreqn, 714
 - ip_msfilter, 715
 - ipc_perm, 512
 - itimerspec, 353
 - itimerval, 326
 - kexec_segment, 282
 - linger, 711
 - mntent, 126
 - mq_attr, 555
 - msgbuf, 519
 - msqid_ds, 518
 - passwd, 274
 - pktinfo, 718
 - pollfd, 437
 - rlimit, 285
 - rusage, 283
 - sched_param, 96
 - sembuf, 531
 - semid_ds, 526
 - semun, 528
 - sem, 527
 - servent, 685
 - shmid_ds, 537
 - sigaction, 335
 - sigevent, 352
 - siginfo_t, 337
 - signalfd_siginfo, 449
 - sigval, 345
 - sock_extended_err, 762
 - sockaddr_atalk, 603
 - sockaddr_in6, 601
 - sockaddr_in, 600
 - sockaddr_ll, 604
 - sockaddr_storage, 600
 - sockaddr_un, 602
 - sockaddr, 599
 - stack_t, 357
 - statfs, 126
 - statx_timestamp, 228
 - statx, 228
 - stat, 151
 - tcp_info, 723
 - termios, 378
 - timespec, 160
 - timeval, 159
 - timex, 297

- tms, 294
- tm, 298
- utimbuf, 158
- utmp, 278
- utsname, 272
- struttura dati del kernel
 - file_lock, 421, 425
 - file_operation, 113
 - file_struct, 61
 - file_system_type, 110
 - file, 112
 - fs_struct, 61
 - inode_operation, 112
 - inode, 111
 - msg, 517
 - msqid_ds, 517
 - sem_queue, 533
 - sem_undo, 533
 - semid_ds, 533
 - sem, 533
 - task_struct, 61
- submount, 120
- suid bit, 164
- superblock, 113
- symbolic link, 129–131
- symlink attack, 147
- SYN flood, 622
- syslog, 369–375
- system call lente, 319–321
- System V Interface Definition (SVID), 14
- TCP window scaling, 742
- termination status, 69–72
- thread, 575–576
- three way handshake, 609–610
- timezone, 295
- tipo
 - elementare, 13
 - opaco, 53–54
 - primitivo, 13
- tipo di dato
 - DIR, 136
 - FILE, 241
 - acl_entry_t, 185
 - acl_type_t, 181
 - acl_t, 179
 - blkcnt_t, 151
 - blksize_t, 151
 - caddr_t, 14
 - cap_flag_t, 402
 - cap_flag_value_t, 403
 - cap_t, 400
 - cap_value_t, 402
 - clock_t, 14
 - clockid_t, 349
 - cpu_set_t, 100
 - dev_t, 14
 - fd_set, 433
 - fpos_t, 247
 - gid_t, 14
 - in_addr_t, 599
 - in_port_t, 599
 - ino_t, 14
 - int16_t, 599
 - int32_t, 599
 - int8_t, 599
 - intptr_t, 38
 - jmp_buf, 55
 - key_t, 14
 - loff_t, 14
 - mode_t, 14
 - mqd_t, 554
 - nfds_t, 437
 - nlink_t, 14
 - off64_t, 21
 - off_t, 14
 - pid_t, 14
 - ptrdiff_t, 14
 - rlim_t, 14
 - sa_family_t, 599
 - sig_atomic_t, 106
 - sighandler_t, 321
 - sigjmp_buf, 358
 - sigset_t, 14
 - sigval_t, 345
 - size_t, 14
 - socklen_t, 599
 - ssize_t, 14
 - tcflag_t, 378
 - time_t, 14
 - timer_t, 352
 - uid_t, 14
 - uint16_t, 599
 - uint32_t, 599
 - uint64_t, 455
 - uint8_t, 599
 - va_list, 53, 54

umask, 168
union filesystem, 226
unix-time, 291
User ID (UID), 81

value result argument, 51
variabili
 automatiche, 34
 di ambiente, 47–50
 globali, 32–33
 statiche, 32–33
Virtual File System (VFS), 8, 109–113

write-behind, 207

zombie, 70–71