

# **GaPiL**

## Guida alla Programmazione in Linux

Simone Piccardi

9 marzo 2005

Copyright © 2000-2004 Simone Piccardi. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 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

<b>Un preambolo</b>	<b>xiii</b>
<b>Prefazione</b>	<b>xv</b>
<b>I Programmazione di sistema</b>	<b>1</b>
<b>1 L'architettura del sistema</b>	<b>3</b>
1.1 Una panoramica . . . . .	3
1.1.1 Concetti base . . . . .	3
1.1.2 User space e kernel space . . . . .	4
1.1.3 Il kernel e il sistema . . . . .	4
1.1.4 Chiamate al sistema e librerie di funzioni . . . . .	5
1.1.5 Un sistema multiutente . . . . .	6
1.2 Gli standard . . . . .	7
1.2.1 Lo standard ANSI C . . . . .	7
1.2.2 I tipi di dati primitivi . . . . .	8
1.2.3 Lo standard IEEE – POSIX . . . . .	8
1.2.4 Lo standard X/Open – XPG3 . . . . .	9
1.2.5 Gli standard Unix – Open Group . . . . .	10
1.2.6 Lo “standard” BSD . . . . .	10
1.2.7 Lo standard System V . . . . .	10
1.2.8 Il comportamento standard del <i>gcc</i> e delle <i>glibc</i> . . . . .	11
<b>2 L'interfaccia base con i processi</b>	<b>13</b>
2.1 Esecuzione e conclusione di un programma . . . . .	13
2.1.1 La funzione <code>main</code> . . . . .	13
2.1.2 Come chiudere un programma . . . . .	14
2.1.3 Le funzioni <code>exit</code> e <code>_exit</code> . . . . .	14
2.1.4 Le funzioni <code>atexit</code> e <code>on_exit</code> . . . . .	15
2.1.5 Conclusioni . . . . .	16
2.2 I processi e l'uso della memoria . . . . .	16
2.2.1 I concetti generali . . . . .	17
2.2.2 La struttura della memoria di un processo . . . . .	17
2.2.3 Allocazione della memoria per i programmi C . . . . .	19
2.2.4 Le funzioni <code>malloc</code> , <code>calloc</code> , <code>realloc</code> e <code>free</code> . . . . .	20
2.2.5 La funzione <code>alloca</code> . . . . .	22
2.2.6 Le funzioni <code>brk</code> e <code>sbrk</code> . . . . .	22
2.2.7 Il controllo della memoria virtuale . . . . .	23
2.3 Argomenti, opzioni ed ambiente di un processo . . . . .	25

2.3.1	Il formato degli argomenti . . . . .	25
2.3.2	La gestione delle opzioni . . . . .	25
2.3.3	Opzioni in formato esteso . . . . .	27
2.3.4	Le variabili di ambiente . . . . .	28
2.4	Problematiche di programmazione generica . . . . .	30
2.4.1	Il passaggio delle variabili e dei valori di ritorno . . . . .	30
2.4.2	Il passaggio di un numero variabile di argomenti . . . . .	31
2.4.3	Potenziali problemi con le variabili automatiche . . . . .	33
2.4.4	Il controllo di flusso non locale . . . . .	33
<b>3</b>	<b>La gestione dei processi</b>	<b>37</b>
3.1	Introduzione . . . . .	37
3.1.1	L'architettura della gestione dei processi . . . . .	37
3.1.2	Una panoramica sulle funzioni fondamentali . . . . .	39
3.2	Le funzioni di base . . . . .	40
3.2.1	Gli identificatori dei processi . . . . .	40
3.2.2	La funzione <code>fork</code> . . . . .	41
3.2.3	La funzione <code>vfork</code> . . . . .	46
3.2.4	La conclusione di un processo. . . . .	46
3.2.5	Le funzioni <code>wait</code> e <code>waitpid</code> . . . . .	48
3.2.6	Le funzioni <code>wait3</code> e <code>wait4</code> . . . . .	51
3.2.7	Le funzioni <code>exec</code> . . . . .	51
3.3	Il controllo di accesso . . . . .	54
3.3.1	Gli identificatori del controllo di accesso . . . . .	54
3.3.2	Le funzioni <code>setuid</code> e <code>setgid</code> . . . . .	56
3.3.3	Le funzioni <code>setreuid</code> e <code>setregid</code> . . . . .	57
3.3.4	Le funzioni <code>seteuid</code> e <code>setegid</code> . . . . .	58
3.3.5	Le funzioni <code>setresuid</code> e <code>setresgid</code> . . . . .	58
3.3.6	Le funzioni <code>setfsuid</code> e <code>setfsgid</code> . . . . .	59
3.3.7	Le funzioni <code>setgroups</code> e <code>getgroups</code> . . . . .	59
3.4	La gestione della priorità di esecuzione . . . . .	61
3.4.1	I meccanismi di <i>scheduling</i> . . . . .	61
3.4.2	Il meccanismo di <i>scheduling standard</i> . . . . .	62
3.4.3	Il meccanismo di <i>scheduling real-time</i> . . . . .	64
3.5	Problematiche di programmazione multitasking . . . . .	67
3.5.1	Le operazioni atomiche . . . . .	67
3.5.2	Le <i>race condition</i> ed i <i>deadlock</i> . . . . .	68
3.5.3	Le funzioni rientranti . . . . .	69
<b>4</b>	<b>L'architettura dei file</b>	<b>71</b>
4.1	L'architettura generale . . . . .	71
4.1.1	L'organizzazione di file e directory . . . . .	71
4.1.2	I tipi di file . . . . .	72
4.1.3	Le due interfacce ai file . . . . .	73
4.2	L'architettura della gestione dei file . . . . .	74
4.2.1	Il <i>Virtual File System</i> di Linux . . . . .	74
4.2.2	Il funzionamento del <i>Virtual File System</i> . . . . .	76
4.2.3	Il funzionamento di un filesystem Unix . . . . .	77
4.2.4	Il filesystem <code>ext2</code> . . . . .	79

<b>5 File e directory</b>	<b>81</b>
5.1 La gestione di file e directory . . . . .	81
5.1.1 Le funzioni <code>link</code> e <code>unlink</code> . . . . .	81
5.1.2 Le funzioni <code>remove</code> e <code>rename</code> . . . . .	83
5.1.3 I link simbolici . . . . .	84
5.1.4 La creazione e la cancellazione delle directory . . . . .	87
5.1.5 La creazione di file speciali . . . . .	88
5.1.6 Accesso alle directory . . . . .	89
5.1.7 La directory di lavoro . . . . .	95
5.1.8 I file temporanei . . . . .	96
5.2 La manipolazione delle caratteristiche dei files . . . . .	98
5.2.1 Le funzioni <code>stat</code> , <code>fstat</code> e <code>lstat</code> . . . . .	99
5.2.2 I tipi di file . . . . .	99
5.2.3 Le dimensioni dei file . . . . .	101
5.2.4 I tempi dei file . . . . .	101
5.2.5 La funzione <code>utime</code> . . . . .	103
5.3 Il controllo di accesso ai file . . . . .	103
5.3.1 I permessi per l'accesso ai file . . . . .	104
5.3.2 I bit <i>suid</i> e <i>sgid</i> . . . . .	106
5.3.3 Il bit <i>sticky</i> . . . . .	107
5.3.4 La titolarità di nuovi file e directory . . . . .	108
5.3.5 La funzione <code>access</code> . . . . .	108
5.3.6 Le funzioni <code>chmod</code> e <code>fchmod</code> . . . . .	109
5.3.7 La funzione <code>umask</code> . . . . .	110
5.3.8 Le funzioni <code>chown</code> , <code>fchown</code> e <code>lchown</code> . . . . .	111
5.3.9 Un quadro d'insieme sui permessi . . . . .	112
5.3.10 La funzione <code>chroot</code> . . . . .	113
<b>6 I file: l'interfaccia standard Unix</b>	<b>115</b>
6.1 L'architettura di base . . . . .	115
6.1.1 L'architettura dei <i>file descriptor</i> . . . . .	115
6.1.2 I file standard . . . . .	116
6.2 Le funzioni base . . . . .	117
6.2.1 La funzione <code>open</code> . . . . .	117
6.2.2 La funzione <code>close</code> . . . . .	120
6.2.3 La funzione <code>lseek</code> . . . . .	120
6.2.4 La funzione <code>read</code> . . . . .	121
6.2.5 La funzione <code>write</code> . . . . .	123
6.3 Caratteristiche avanzate . . . . .	123
6.3.1 La condivisione dei files . . . . .	123
6.3.2 Operazioni atomiche con i file . . . . .	125
6.3.3 Le funzioni <code>sync</code> e <code>fsync</code> . . . . .	126
6.3.4 Le funzioni <code>dup</code> e <code>dup2</code> . . . . .	127
6.3.5 La funzione <code>fcntl</code> . . . . .	129
6.3.6 La funzione <code>ioctl</code> . . . . .	131
<b>7 I file: l'interfaccia standard ANSI C</b>	<b>133</b>
7.1 Introduzione . . . . .	133
7.1.1 I <i>file stream</i> . . . . .	133
7.1.2 Gli oggetti <code>FILE</code> . . . . .	134
7.1.3 Gli stream standard . . . . .	134

7.1.4	Le modalità di bufferizzazione . . . . .	134
7.2	Funzioni base . . . . .	135
7.2.1	Apertura e chiusura di uno stream . . . . .	136
7.2.2	Lettura e scrittura su uno stream . . . . .	138
7.2.3	Input/output binario . . . . .	139
7.2.4	Input/output a caratteri . . . . .	140
7.2.5	Input/output di linea . . . . .	142
7.2.6	L'input/output formattato . . . . .	144
7.2.7	Posizionamento su uno stream . . . . .	148
7.3	Funzioni avanzate . . . . .	149
7.3.1	Le funzioni di controllo . . . . .	149
7.3.2	Il controllo della bufferizzazione . . . . .	150
7.3.3	Gli stream e i thread . . . . .	152
<b>8</b>	<b>La gestione del sistema, del tempo e degli errori</b>	<b>155</b>
8.1	Capacità e caratteristiche del sistema . . . . .	155
8.1.1	Limiti e parametri di sistema . . . . .	155
8.1.2	La funzione <code>sysconf</code> . . . . .	158
8.1.3	I limiti dei file . . . . .	159
8.1.4	La funzione <code>pathconf</code> . . . . .	160
8.1.5	La funzione <code>uname</code> . . . . .	160
8.2	Opzioni e configurazione del sistema . . . . .	161
8.2.1	La funzione <code>sysctl</code> ed il filesystem <code>/proc</code> . . . . .	161
8.2.2	La gestione delle proprietà dei filesystem . . . . .	163
8.2.3	La gestione delle informazioni su utenti e gruppi . . . . .	166
8.2.4	Il registro della <i>contabilità</i> degli utenti . . . . .	169
8.3	Limitazione ed uso delle risorse . . . . .	171
8.3.1	L'uso delle risorse . . . . .	171
8.3.2	Limiti sulle risorse . . . . .	172
8.3.3	Le risorse di memoria e processore . . . . .	173
8.4	La gestione dei tempi del sistema . . . . .	175
8.4.1	La misura del tempo in Unix . . . . .	175
8.4.2	La gestione del <i>process time</i> . . . . .	176
8.4.3	Le funzioni per il <i>calendar time</i> . . . . .	177
8.4.4	La gestione delle date. . . . .	181
8.5	La gestione degli errori . . . . .	183
8.5.1	La variabile <code>errno</code> . . . . .	184
8.5.2	Le funzioni <code>strerror</code> e <code>perror</code> . . . . .	184
8.5.3	Alcune estensioni GNU . . . . .	185
<b>9</b>	<b>I segnali</b>	<b>189</b>
9.1	Introduzione . . . . .	189
9.1.1	I concetti base . . . . .	189
9.1.2	Le <i>semantiche</i> del funzionamento dei segnali . . . . .	190
9.1.3	Tipi di segnali . . . . .	191
9.1.4	La notifica dei segnali . . . . .	191
9.2	La classificazione dei segnali . . . . .	192
9.2.1	I segnali standard . . . . .	192
9.2.2	Segnali di errore di programma . . . . .	193
9.2.3	I segnali di terminazione . . . . .	195
9.2.4	I segnali di allarme . . . . .	196

9.2.5	I segnali di I/O asincrono . . . . .	196
9.2.6	I segnali per il controllo di sessione . . . . .	197
9.2.7	I segnali di operazioni errate . . . . .	197
9.2.8	Ulteriori segnali . . . . .	198
9.2.9	Le funzioni <code>strsignal</code> e <code>psignal</code> . . . . .	198
9.3	La gestione dei segnali . . . . .	199
9.3.1	Il comportamento generale del sistema. . . . .	199
9.3.2	La funzione <code>signal</code> . . . . .	200
9.3.3	Le funzioni <code>kill</code> e <code>raise</code> . . . . .	201
9.3.4	Le funzioni <code>alarm</code> e <code>abort</code> . . . . .	203
9.3.5	Le funzioni di pausa e attesa . . . . .	206
9.3.6	Un esempio elementare . . . . .	207
9.4	Gestione avanzata . . . . .	208
9.4.1	Alcune problematiche aperte . . . . .	209
9.4.2	Gli <i>insiemi di segnali</i> o <i>signal set</i> . . . . .	211
9.4.3	La funzione <code>sigaction</code> . . . . .	212
9.4.4	La gestione della <i>maschera dei segnali</i> o <i>signal mask</i> . . . . .	215
9.4.5	Ulteriori funzioni di gestione . . . . .	217
9.4.6	I segnali real-time . . . . .	220
<b>10</b>	<b>Terminali e sessioni di lavoro</b>	<b>223</b>
10.1	Il <i>job control</i> . . . . .	223
10.1.1	Una panoramica introduttiva . . . . .	223
10.1.2	I <i>process group</i> e le <i>sessions</i> . . . . .	224
10.1.3	Il terminale di controllo e il controllo di sessione . . . . .	226
10.1.4	Dal login alla shell . . . . .	229
10.1.5	Prescrizioni per un programma <i>daemon</i> . . . . .	230
10.2	L'I/O su terminale . . . . .	234
10.2.1	L'architettura . . . . .	235
10.2.2	La gestione delle caratteristiche di un terminale . . . . .	236
10.2.3	La gestione della disciplina di linea. . . . .	246
10.2.4	Operare in <i>modo non canonico</i> . . . . .	247
10.3	La gestione dei terminali virtuali . . . . .	248
10.3.1	I terminali virtuali . . . . .	248
10.3.2	La funzione <code>openpty</code> . . . . .	248
<b>11</b>	<b>La gestione avanzata dei file</b>	<b>249</b>
11.1	L' <i>I/O multiplexing</i> . . . . .	249
11.1.1	La problematica dell' <i>I/O multiplexing</i> . . . . .	249
11.1.2	Le funzioni <code>select</code> e <code>pselect</code> . . . . .	250
11.1.3	La funzione <code>poll</code> . . . . .	253
11.2	L'accesso asincrono ai file . . . . .	254
11.2.1	Operazioni asincrone sui file . . . . .	255
11.2.2	L'interfaccia POSIX per l'I/O asincrono . . . . .	256
11.3	Altre modalità di I/O avanzato . . . . .	260
11.3.1	I/O vettorizzato . . . . .	261
11.3.2	File mappati in memoria . . . . .	261
11.4	Il file locking . . . . .	267
11.4.1	L' <i>advisory locking</i> . . . . .	267
11.4.2	La funzione <code>flock</code> . . . . .	268
11.4.3	Il file locking POSIX . . . . .	270

11.4.4 La funzione <code>lockf</code> . . . . .	276
11.4.5 Il <i>mandatory locking</i> . . . . .	277
<b>12 La comunicazione fra processi</b>	<b>279</b>
12.1 La comunicazione fra processi tradizionale . . . . .	279
12.1.1 Le <i>pipe</i> standard . . . . .	279
12.1.2 Un esempio dell'uso delle pipe . . . . .	281
12.1.3 Le funzioni <code>popen</code> e <code>pclose</code> . . . . .	283
12.1.4 Le <i>pipe</i> con nome, o <i>fifo</i> . . . . .	286
12.1.5 La funzione <code>socketpair</code> . . . . .	291
12.2 Il sistema di comunicazione fra processi di System V . . . . .	292
12.2.1 Considerazioni generali . . . . .	292
12.2.2 Il controllo di accesso . . . . .	294
12.2.3 Gli identificatori ed il loro utilizzo . . . . .	295
12.2.4 Code di messaggi . . . . .	297
12.2.5 Semafori . . . . .	306
12.2.6 Memoria condivisa . . . . .	316
12.3 Tecniche alternative . . . . .	327
12.3.1 Alternative alle code di messaggi . . . . .	327
12.3.2 I <i>file di lock</i> . . . . .	327
12.3.3 La sincronizzazione con il <i>file locking</i> . . . . .	329
12.3.4 Il <i>memory mapping</i> anonimo . . . . .	331
12.4 Il sistema di comunicazione fra processi di POSIX . . . . .	331
12.4.1 Considerazioni generali . . . . .	331
12.4.2 Code di messaggi . . . . .	332
12.4.3 Semafori . . . . .	338
12.4.4 Memoria condivisa . . . . .	338
<b>II Programmazione di rete</b>	<b>343</b>
<b>13 Introduzione alla programmazione di rete</b>	<b>345</b>
13.1 Modelli di programmazione . . . . .	345
13.1.1 Il modello <i>client-server</i> . . . . .	345
13.1.2 Il modello <i>peer-to-peer</i> . . . . .	346
13.1.3 Il modello <i>three-tier</i> . . . . .	346
13.2 I protocolli di rete . . . . .	347
13.2.1 Il modello ISO/OSI . . . . .	347
13.2.2 Il modello TCP/IP (o DoD) . . . . .	348
13.2.3 Criteri generali dell'architettura del TCP/IP . . . . .	350
13.3 Il protocollo TCP/IP . . . . .	350
13.3.1 Il quadro generale . . . . .	351
13.3.2 Internet Protocol (IP) . . . . .	353
13.3.3 User Datagram Protocol (UDP) . . . . .	354
13.3.4 Transport Control Protocol (TCP) . . . . .	354
13.3.5 Limiti e dimensioni riguardanti la trasmissione dei dati . . . . .	355

<b>14 Introduzione ai socket</b>	<b>357</b>
14.1 Una panoramica . . . . .	357
14.1.1 I <i>socket</i> . . . . .	357
14.1.2 Concetti base . . . . .	357
14.2 La creazione di un <i>socket</i> . . . . .	358
14.2.1 La funzione <b>socket</b> . . . . .	358
14.2.2 Il dominio, o <i>protocol family</i> . . . . .	359
14.2.3 Il tipo, o stile . . . . .	360
14.3 Le strutture degli indirizzi dei socket . . . . .	361
14.3.1 La struttura generica . . . . .	361
14.3.2 La struttura degli indirizzi IPv4 . . . . .	362
14.3.3 La struttura degli indirizzi IPv6 . . . . .	363
14.3.4 La struttura degli indirizzi locali . . . . .	363
14.3.5 La struttura degli indirizzi AppleTalk . . . . .	364
14.3.6 La struttura degli indirizzi dei <i>packet socket</i> . . . . .	365
14.4 Le funzioni di conversione degli indirizzi . . . . .	366
14.4.1 La <i>endianess</i> . . . . .	366
14.4.2 Le funzioni per il riordinamento . . . . .	368
14.4.3 Le funzioni <b>inet_aton</b> , <b>inet_addr</b> e <b>inet_ntoa</b> . . . . .	368
14.4.4 Le funzioni <b>inet_pton</b> e <b>inet_ntop</b> . . . . .	369
<b>15 I socket TCP</b>	<b>371</b>
15.1 Il funzionamento di una connessione TCP . . . . .	371
15.1.1 La creazione della connessione: il <i>three way handshake</i> . . . . .	371
15.1.2 Le opzioni TCP . . . . .	372
15.1.3 La terminazione della connessione . . . . .	373
15.1.4 Un esempio di connessione . . . . .	374
15.1.5 Lo stato <b>TIME_WAIT</b> . . . . .	376
15.1.6 I numeri di porta . . . . .	377
15.1.7 Le porte ed il modello client/server . . . . .	379
15.2 Le funzioni di base per la gestione dei socket . . . . .	380
15.2.1 La funzione <b>bind</b> . . . . .	380
15.2.2 La funzione <b>connect</b> . . . . .	382
15.2.3 La funzione <b>listen</b> . . . . .	383
15.2.4 La funzione <b>accept</b> . . . . .	385
15.2.5 Le funzioni <b>getsockname</b> e <b>getpeername</b> . . . . .	386
15.2.6 La funzione <b>close</b> . . . . .	387
15.3 Un esempio elementare: il servizio <i>daytime</i> . . . . .	388
15.3.1 Il comportamento delle funzioni di I/O . . . . .	388
15.3.2 Il client <i>daytime</i> . . . . .	389
15.3.3 Un server <i>daytime</i> iterativo . . . . .	392
15.3.4 Un server <i>daytime</i> concorrente . . . . .	394
15.4 Un esempio più completo: il servizio <i>echo</i> . . . . .	396
15.4.1 Il servizio <i>echo</i> . . . . .	396
15.4.2 Il client <i>echo</i> : prima versione . . . . .	396
15.4.3 Il server <i>echo</i> : prima versione . . . . .	398
15.4.4 L'avvio e il funzionamento normale . . . . .	400
15.4.5 La conclusione normale . . . . .	402
15.4.6 La gestione dei processi figli . . . . .	403
15.5 I vari scenari critici . . . . .	407
15.5.1 La terminazione precoce della connessione . . . . .	407

15.5.2 La terminazione precoce del server . . . . .	408
15.5.3 Altri scenari di terminazione della connessione . . . . .	411
15.6 L'uso dell'I/O multiplexing . . . . .	415
15.6.1 Il comportamento della funzione <code>select</code> con i socket. . . . .	415
15.6.2 Un esempio di I/O multiplexing . . . . .	416
15.6.3 La funzione <code>shutdown</code> . . . . .	419
15.6.4 Un server basato sull'I/O multiplexing . . . . .	422
15.6.5 I/O multiplexing con <code>poll</code> . . . . .	426
<b>16 La gestione dei socket</b>	<b>431</b>
16.1 La risoluzione dei nomi . . . . .	431
16.1.1 La struttura del <i>resolver</i> . . . . .	431
16.1.2 Le funzioni di interrogazione del <i>resolver</i> . . . . .	433
16.1.3 La risoluzione dei nomi a dominio . . . . .	438
16.1.4 Le funzioni avanzate per la risoluzione dei nomi . . . . .	446
16.2 Le opzioni dei socket . . . . .	456
16.2.1 Le funzioni <code>setsockopt</code> e <code>getsockopt</code> . . . . .	457
16.2.2 Le opzioni generiche . . . . .	458
16.3 Altre funzioni di controllo . . . . .	463
16.3.1 L'uso di <code>fcntl</code> per i socket . . . . .	463
16.3.2 L'uso di <code>ioctl</code> per i socket . . . . .	463
16.3.3 L'uso di <code>sysctl</code> per le proprietà della rete . . . . .	463
<b>17 Gli altri tipi di socket</b>	<b>465</b>
17.1 I socket UDP . . . . .	465
17.1.1 Le caratteristiche di un socket UDP . . . . .	465
17.1.2 Le funzioni <code>sendto</code> e <code>recvfrom</code> . . . . .	466
17.1.3 Un client UDP elementare . . . . .	469
17.1.4 Un server UDP elementare . . . . .	470
17.1.5 Le problematiche dei socket UDP . . . . .	472
17.1.6 L'uso della funzione <code>connect</code> con i socket UDP . . . . .	476
17.2 I socket <i>Unix domain</i> . . . . .	477
17.3 Altri socket . . . . .	477
<b>18 Socket avanzati</b>	<b>479</b>
18.1 Le funzioni di I/O avanzate . . . . .	479
18.1.1 I dati <i>out-of-band</i> . . . . .	479
18.2 L'uso dell'I/O non bloccante . . . . .	479
<b>III Appendici</b>	<b>481</b>
<b>A Il livello di rete</b>	<b>483</b>
A.1 Il protocollo IP . . . . .	483
A.1.1 Introduzione . . . . .	483
A.2 Il protocollo IPv6 . . . . .	484
A.2.1 I motivi della transizione . . . . .	485
A.2.2 Principali caratteristiche di IPv6 . . . . .	485
A.2.3 L'intestazione di IPv6 . . . . .	486
A.2.4 Gli indirizzi di IPv6 . . . . .	488
A.2.5 La notazione . . . . .	488

A.2.6	La architettura degli indirizzi di IPv6 . . . . .	489
A.2.7	Indirizzi unicast <i>provider-based</i> . . . . .	490
A.2.8	Indirizzi ad uso locale . . . . .	491
A.2.9	Indirizzi riservati . . . . .	492
A.2.10	Multicasting . . . . .	492
A.2.11	Indirizzi <i>anycast</i> . . . . .	493
A.2.12	Le estensioni . . . . .	494
A.2.13	Qualità di servizio . . . . .	494
A.2.14	Etichette di flusso . . . . .	495
A.2.15	Priorità . . . . .	495
A.2.16	Sicurezza a livello IP . . . . .	496
A.2.17	Autenticazione . . . . .	496
A.2.18	Riservatezza . . . . .	497
A.2.19	Autoconfigurazione . . . . .	498
A.2.20	Autoconfigurazione stateless . . . . .	498
A.2.21	Autoconfigurazione stateful . . . . .	499
<b>B</b>	<b>Il livello di trasporto</b>	<b>501</b>
B.1	Il protocollo TCP . . . . .	501
B.1.1	Gli stati del TCP . . . . .	501
B.2	Il protocollo UDP . . . . .	501
<b>C</b>	<b>I codici di errore</b>	<b>503</b>
C.1	Gli errori dei file . . . . .	503
C.2	Gli errori dei processi . . . . .	505
C.3	Gli errori di rete . . . . .	505
C.4	Errori generici . . . . .	507
C.5	Errori del kernel . . . . .	509
<b>D</b>	<b>Ringraziamenti</b>	<b>511</b>
<b>E</b>	<b>GNU Free Documentation License</b>	<b>513</b>
E.1	Applicability and Definitions . . . . .	513
E.2	Verbatim Copying . . . . .	514
E.3	Copying in Quantity . . . . .	514
E.4	Modifications . . . . .	515
E.5	Combining Documents . . . . .	516
E.6	Collections of Documents . . . . .	517
E.7	Aggregation With Independent Works . . . . .	517
E.8	Translation . . . . .	517
E.9	Termination . . . . .	517
E.10	Future Revisions of This License . . . . .	517



# Un preambolo

Questa guida nasce dalla mia profonda convizione 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, encyclopedie, 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, etc.

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 le *GNU C Library* o *glibc*, che ormai sono usate nella stragrande maggioranza dei casi.

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 [1] e [2]) 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 L<sup>A</sup>T<sub>E</sub>X 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 del sistema. Chi avesse già una conoscenza di questa materia può tranquillamente saltare questa sezione.

#### 1.1.1 Concetti base

Il concetto base 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 essenziali (la CPU, la memoria, le periferiche) mentre tutto il resto, quindi anche la parte che prevede l’interazione con l’utente, deve venire realizzato tramite programmi eseguiti dal kernel e che accedano alle risorse hardware tramite delle richieste a quest’ultimo.

Fin dall’inizio uno Unix si presenta come un sistema operativo *multitasking*, cioè in grado di eseguire contemporaneamente più programmi, e multutente, in cui è possibile che più utenti siano connessi ad una macchina eseguendo più programmi “in contemporanea” (in realtà, almeno per macchine a processore singolo, i programmi vengono eseguiti singolarmente a rotazione).

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 ad essere eseguito in modalità privilegiata, con il completo accesso all’hardware, mentre i programmi normali vengono eseguiti in modalità protetta (e non possono accedere direttamente alle zone di memoria riservate o alle porte di input/output).

Una parte del kernel, lo *scheduler*, si occupa di stabilire, ad intervalli fissi e sulla base di un opportuno calcolo delle priorità, quale “processo” deve essere posto in esecuzione (il cosiddetto *preemptive scheduling*). Questo verrà comunque eseguito in modalità protetta; quando necessario il processo potrà accedere alle risorse hardware soltanto attraverso delle opportune chiamate al sistema che restituiranno il controllo al kernel.

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 disponibile, salvando su disco quando necessario (nella cosiddetta area di *swap*) le pagine di memoria in eccedenza.

Le periferiche infine vengono viste in genere attraverso un’interfaccia astratta che permette di trattarle come fossero file, secondo il concetto per cui *everything is a file*, su cui torneremo in dettaglio in cap. 4, (questo non è vero per le interfacce di rete, che hanno un’interfaccia diversa, ma resta valido il concetto generale che tutto il lavoro di accesso e gestione a basso livello è effettuato dal kernel).

### 1.1.2 User space e kernel space

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 dall’architettura, 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 sistema 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 che vengono per vari motivi eseguiti al livello del kernel.

Pertanto deve essere chiaro a chi programma in Unix 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 allo user space.

### 1.1.3 Il kernel e il sistema

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’avvio il BIOS (o in generale il software di avvio posto nelle EPROM) eseguirà la procedura di avvio del sistema (il cosiddetto *boot*), incaricandosi di caricare il kernel in memoria e di farne partire l’esecuzione; quest’ultimo, dopo aver inizializzato le periferiche, farà partire il primo processo, *init*, che è quello che a sua volta farà partire tutti i processi successivi. Fra questi ci sarà pure quello che si occupa di dialogare con la tastiera e lo schermo della console, e quello che mette a disposizione dell’utente che si vuole collegare, un terminale e la *shell* da cui inviare i comandi.

E’ da rimarcare come tutto ciò, 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, in user space.

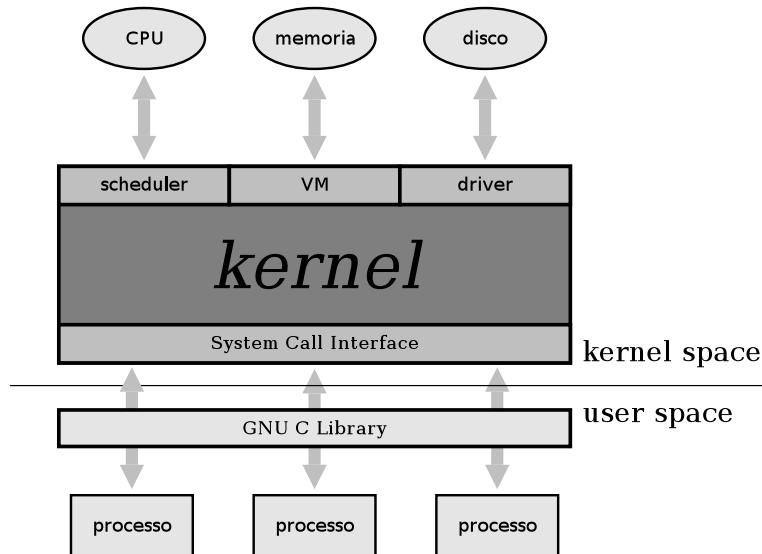
Questo significa, ad esempio, che il sistema di per sé non dispone di primitive per tutta una serie di operazioni (come la copia di un file) che altri sistemi (come Windows) hanno invece al loro interno. Pertanto buona parte delle operazioni di normale amministrazione di un sistema, come quella in esempio, sono implementate come normali programmi.

Per questo motivo quando ci si riferisce al sistema nella sua interezza è corretto parlare di un sistema GNU/Linux: da solo il kernel è assolutamente inutile; quello che costruisce un sistema operativo utilizzabile è la presenza di tutta una serie di librerie e programmi di utilità (che di norma sono quelli realizzati dal progetto GNU della Free Software Foundation) che permettono di eseguire le normali operazioni che ci si aspetta da un sistema operativo.

### 1.1.4 Chiamate al sistema e librerie di funzioni

Come accennato le interfacce con cui i programmi possono accedere all'hardware vanno sotto il nome di chiamate al sistema (le cosiddette *system call*), si tratta di un insieme di funzioni che un programma può chiamare, per le quali viene generata un'interruzione del processo passando il controllo dal programma al kernel. Sarà poi quest'ultimo che (oltre a compiere una serie di operazioni interne come la gestione del multitasking e l'allocazione della memoria) eseguirà la funzione richiesta in *kernel space* restituendo i risultati al chiamante.

Ogni versione di Unix ha storicamente sempre avuto un certo numero di queste chiamate, che sono riportate nella seconda sezione del *Manuale di programmazione di Unix* (quella cui si accede con il comando `man 2 <nome>`) e Linux non fa eccezione. Queste sono poi state codificate da vari standard, che esamineremo brevemente in sez. 1.2. 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.

Normalmente ciascuna di queste chiamate al sistema viene rimappata in opportune funzioni con lo stesso nome definite dentro la Libreria Standard del C, che, oltre alle interfacce alle system call, contiene anche tutta la serie delle ulteriori funzioni definite dai vari standard, che sono comunemente usate nella programmazione.

Questo è importante da capire perché programmare in Linux significa anzitutto essere in grado di usare le varie interfacce contenute nella Libreria Standard del C, in quanto né il kernel, né il linguaggio C, implementano direttamente operazioni comuni come l'allocazione dinamica della memoria, l'input/output bufferizzato o la manipolazione delle stringhe, presenti in qualunque programma.

Quanto appena illustrato mette in evidenza il fatto che nella stragrande maggioranza dei casi,<sup>1</sup> si dovrebbe usare il nome GNU/Linux (piuttosto che soltanto Linux) in quanto una parte essenziale del sistema (senza la quale niente funzionerebbe) è la GNU Standard C Library (in breve *glibc*), ovvero la libreria realizzata dalla Free Software Foundation nella quale sono state

<sup>1</sup>esistono implementazioni diverse delle librerie Standard del C, come le *libc5* o le *uClib*, che non derivano dal progetto GNU. Le *libc5* oggi sono, tranne casi particolari, completamente soppiantate dalle *glibc*, le *uClib* pur non essendo complete come le *glibc*, restano invece molto diffuse nel mondo embedded per le loro dimensioni ridotte (e soprattutto la possibilità di togliere le parti non necessarie), e pertanto costituiscono un valido rimpiazzo delle *glibc* in tutti quei sistemi specializzati che richiedono una minima occupazione di memoria.

implementate tutte le funzioni essenziali definite negli standard POSIX e ANSI C, utilizzabili da qualunque programma.

Le funzioni di questa libreria sono quelle riportate dalla terza sezione del *Manuale di Programmazione di Unix* (cioè accessibili con il comando `man 3 <nome>`) e sono costruite sulla base delle chiamate al sistema del kernel; è importante avere presente questa distinzione, fondamentale dal punto di vista dell'implementazione, anche se poi, nella realizzazione di normali programmi, non si hanno differenze pratiche fra l'uso di una funzione di libreria e quello di una chiamata al sistema.

### 1.1.5 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 presente.

Il concetto base è quello di utente (*user*) del sistema, le cui capacità rispetto a quello che può fare sono sottoposte a ben precisi limiti. 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.

Ogni utente è identificato da un nome (l'*username*), che è quello che viene richiesto all'ingresso nel sistema dalla procedura di *login* (descritta in dettaglio in sez. 10.1.4). 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.<sup>2</sup>

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, etc.

L'utente e il gruppo sono identificati da due numeri (la cui corrispondenza ad un nome espresso in caratteri è inserita nei due file `/etc/passwd` e `/etc/groups`). Questi numeri 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*, e sono quelli che vengono usati dal kernel per identificare l'utente.

In questo modo 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. 5.3) è regolato da questo meccanismo di identificazione.

Infine in ogni Unix è presente un utente speciale 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 descritti in precedenza sono disattivati.<sup>3</sup>

---

<sup>2</sup>Ad esempio usando la libreria PAM (*Pluggable Authentication Methods*) è possibile astrarre completamente dai meccanismi di autenticazione e sostituire ad esempio l'uso delle password con meccanismi di identificazione biometrica.

<sup>3</sup>i controlli infatti vengono sempre eseguiti da un codice del tipo `if (uid) { ... }`

## 1.2 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 del sistema o alle caratteristiche che si sono stabilite come standard di fatto in quanto facenti parte di alcune implementazioni molto diffuse come BSD o SVr4.

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 modo essi sono supportati sia per quanto riguarda il kernel che le librerie del C (con una particolare attenzione alle *glibc*).

### 1.2.1 Lo standard ANSI C

Lo standard ANSI C è stato definito nel 1989 dall'*American National Standard Institute*, come standard del linguaggio C ed è stato successivamente adottato dalla *International Standard Organisation* come standard internazionale con la sigla ISO/IEC 9899:1990, e va anche sotto il nome di standard ISO C.

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 le *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*<sup>4</sup> (anch'essi provvisti dalla *glibc*). In tab. 1.1 si sono riportati i principali *header file* definiti nello standard POSIX, insieme a quelli definiti negli altri standard descritti nelle sezioni successive.

Header	Standard		Contenuto
	ANSI C	POSIX	
<b>assert.h</b>	•	•	Verifica le asserzioni fatte in un programma.
<b>errno.h</b>	•	•	Errori di sistema.
<b>fcntl.h</b>	•	•	Controllo sulle opzioni dei file.
<b>limits.h</b>	•	•	Limits e parametri del sistema.
	•	•	.
	•	•	.
	•	•	.
	•	•	.
<b>stdio.h</b>	•	•	I/O bufferizzato in standard ANSI C.
<b>stdlib.h</b>	•	•	definizioni della libreria standard.

**Tabella 1.1:** Elenco dei vari header file definiti dallo standard POSIX.

In realtà *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 addizionali) 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.

<sup>4</sup>i file di dichiarazione di variabili, tipi e funzioni, usati normalmente da un compilatore C. Per poter accedere alle funzioni occorre includere con la direttiva `#include` questi file nei propri programmi; per ciascuna funzione che tratteremo in seguito indicheremo anche gli *header file* necessari ad usarla.

### 1.2.2 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 sempre stata associata ad un intero a 32 bit, mentre il numero di dispositivo è sempre 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.

Tipo	Contenuto
<code>caddr_t</code>	core address.
<code>clock_t</code>	contatore del tempo di sistema.
<code>dev_t</code>	Numero di dispositivo.
<code>gid_t</code>	Identificatore di un gruppo.
<code>ino_t</code>	Numero di <i>inode</i> .
<code>key_t</code>	Chiave per il System V IPC.
<code>loff_t</code>	Posizione corrente in un file.
<code>mode_t</code>	Attributi di un file.
<code>nlink_t</code>	Contattore dei link su un file.
<code>off_t</code>	Posizione corrente in un file.
<code>pid_t</code>	Identificatore di un processo.
<code>rlim_t</code>	Limite sulle risorse.
<code>sigset_t</code>	Insieme di segnali.
<code>size_t</code>	Dimensione di un oggetto.
<code>ssize_t</code>	Dimensione in numero di byte ritornata dalle funzioni.
<code>ptrdiff_t</code>	Differenza fra due puntatori.
<code>time_t</code>	Numero di secondi (in tempo di calendario).
<code>uid_t</code>	Identificatore di un utente.

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

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.

### 1.2.3 Lo standard IEEE – POSIX

Uno standard più attinente al sistema nel suo complesso (e che concerne sia il kernel che le librerie) è 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 realtime e per i thread (1003.1d e 1003.1c) 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 si può 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; una pagina dove si possono recuperare varie (e di norma piuttosto intricate) informazioni è: <http://www.pasc.org/standing/sd11.html>.

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). Lo standard principale resta comunque POSIX.1, che continua ad evolversi; la versione più nota, cui gran parte delle implementazioni fanno riferimento, e che costituisce una base per molti altri tentativi di standardizzazione, è stata rilasciata anche come standard internazionale con la sigla ISO/IEC 9945-1:1996.

Linux e le *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. ??), e dallo standard POSIX.1b per quanto riguarda i segnali e lo scheduling real-time (sez. 9.4.6 e sez. 3.4.3), la misura del tempo, i meccanismi di intercomunicazione (sez. 12.4) e l'I/O asincrono (sez. 11.2.2).

#### 1.2.4 Lo standard X/Open – XPG3

Il consorzio X/Open nacque nel 1984 come consorzio di vendori 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 XPGn).

Nel 1989 produsse una terza versione di questa guida particolarmente voluminosa (la *X/Open Portability Guide, Issue 3*), contenente un'ulteriore 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 (*Application Programmable Interface*) 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, header e comandi definiti).

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*, l’ultima versione di Spec 1170 diventò così la prima versione delle *Single UNIX Specification*, SUSv1, più comunemente nota come *Unix 95*.

### 1.2.5 Gli standard Unix – Open Group

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 queste versione le interfacce specificate salgono a 1434 (e 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.

### 1.2.6 Lo “standard” BSD

Lo sviluppo di BSD iniziò quando la fine della collaborazione fra l’Università di Berkley e la AT&T generò una delle prime e più importanti fratture del mondo Unix. L’Università di Berkley proseguì nello sviluppo della base di codice di cui disponeva, e che presentava parecchie migliorie rispetto alle allora versioni 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 uno standard formalizzato, l’implementazione di Unix dell’Università di Berkley, ha provveduto nel tempo una serie di estensioni e API di grande rilievo, come il link simbolici, la funzione `select`, i socket.

Queste estensioni sono state via via aggiunte al sistema nelle varie versioni del sistema (BSD 4.2, BSD 4.3 e BSD 4.4) come pure in alcuni derivati commerciali come SunOS. Il kernel e le `glibc` provvedono tutte queste estensioni che sono state in gran parte incorporate negli standard successivi.

### 1.2.7 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. Negli anni successivi l’AT&T proseguì lo sviluppo rilasciando varie versioni con aggiunte e integrazioni; nel 1989 un accordo fra vari vendori (AT&T, Sun, HP, e altro) portò ad una versione che provvedeva un’unificazione delle interfacce comprendente Xenix e BSD, la System V release 4.

L’interfaccia di questa ultima release è descritta in un documento dal titolo *System V Interface Description*, o SVID; spesso però si fa riferimento a questo standard con il nome della sua implementazione, usando la sigla SVr4.

Anche questo 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 93.

Linux e le *glibc* implementano le principali funzionalità richieste da 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. 12.2) le funzioni della famiglia `hsearch` e `drand48`, `fmtmsg` e svariate funzioni matematiche.

### 1.2.8 Il comportamento standard del `gcc` e delle *glibc*

In Linux, grazie alle *glibc*, gli standard appena descritti sono ottenibili sia attraverso l'uso di opzioni del compilatore (il `gcc`) che definendo opportune costanti prima dell'inclusione dei file degli header.

Se si vuole che i programmi seguano una stretta attinenza allo standard ANSI C si può usare l'opzione `-ansi` del compilatore, e non sarà riconosciuta nessuna funzione non riconosciuta dalle specifiche standard ISO per il C.

Per attivare le varie opzioni è possibile definire le macro di preprocessore, che controllano le funzionalità che le *glibc* possono mettere a disposizione: questo può essere fatto attraverso l'opzione `-D` del compilatore, ma è buona norma inserire gli opportuni `#define` nei propri header file.

Le macro disponibili per i vari standard sono le seguenti:

`_POSIX_SOURCE` definendo questa macro 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. Se è uguale a '1' vengono attivate le funzionalità specificate nella edizione del 1990 (IEEE Standard 1003.1-1990), valori maggiori o uguali a '2' attivano le funzionalità POSIX.2 specificate nell'edizione del 1992 (IEEE Standard 1003.2-1992). Un valore maggiore o uguale a '199309L' attiva le funzionalità POSIX.1b specificate nell'edizione del 1993 (IEEE Standard 1003.1b-1993). Un valore maggiore o uguale a '199506L' attiva le funzionalità POSIX.1 specificate nell'edizione del 1996 (ISO/IEC 9945-1: 1996). Valori superiori abiliteranno ulteriori estensioni.

`_BSD_SOURCE` definendo questa macro si attivano 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 le definizioni previste da BSD4.3 hanno la precedenza rispetto a POSIX. A causa della natura dei conflitti con POSIX per ottenere una piena compatibilità con BSD4.3 è necessario anche usare una libreria di compatibilità, dato che alcune funzioni sono definite in

modo diverso. In questo caso occorre pertanto anche usare l'opzione `-lbsd-compat` con il compilatore per indicargli di utilizzare le versioni nella libreria di compatibilità prima di quelle normali.

`_SVID_SOURCE` definendo questa macro si attivano le funzionalità derivate da SVID. Esse comprendono anche quelle definite negli standard ISO C, POSIX.1, POSIX.2, e X/Open.

`_XOPEN_SOURCE` definendo questa macro si attivano le funzionalità descritte nella *X/Open Portability Guide*. Anche queste sono un sovrainsieme di quelle definite in 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. Se il valore della macro è posto a 500 questo include anche le nuove definizioni introdotte con la *Single UNIX Specification, version 2*, cioè Unix98.

`_XOPEN_SOURCE_EXTENDED` definendo questa macro si attivano le ulteriori funzionalità necessarie ad essere conformi al rilascio del marchio *X/Open Unix*.

`_ISO_C99_SOURCE` definendo questa macro si attivano le funzionalità previste per la revisione delle librerie standard del C denominato ISO C99. Dato che lo standard non è ancora adottato in maniera ampia queste non sono abilitate automaticamente, ma le *glibc* hanno già un'implementazione completa che può essere attivata definendo questa macro.

`_LARGEFILE_SOURCE` definendo questa macro si attivano le funzionalità per il supporto dei file di grandi dimensioni (il *Large File Support* o LFS) con indici e dimensioni a 64 bit.

`_GNU_SOURCE` definendo questa macro si attivano tutte le funzionalità disponibili: ISO C89, ISO C99, POSIX.1, POSIX.2, BSD, SVID, X/Open, LFS più le estensioni specifiche GNU. Nel caso in cui BSD e POSIX confliggano viene data la precedenza a POSIX.

In particolare è da sottolineare che le *glibc* supportano alcune estensioni specifiche GNU, che non sono comprese in nessuno degli standard citati. Per poterle utilizzare esse devono essere attivate esplicitamente definendo la macro `_GNU_SOURCE` prima di includere i vari header file.

# 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.<sup>1</sup>

#### 2.1.1 La funzione `main`

Quando un programma viene lanciato il kernel esegue un’opportuna routine di avvio, usando il programma `ld-linux.so`. Questo programma prima carica le librerie condivise che servono al programma, poi effettua il link dinamico del codice e alla fine lo esegue. Infatti, a meno di non aver specificato il flag `-static` durante la compilazione, tutti i programmi in Linux sono incompleti e necessitano di essere *linkati* alle librerie condivise quando vengono avviati. La procedura è controllata da alcune variabili di ambiente e dal contenuto di `/etc/ld.so.conf`. I dettagli sono riportati nella man page di `ld.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 iniziare l’esecuzione; in ogni caso senza questa funzione lo stesso *linker* 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, in sostanza un prototipo che va sempre bene è il seguente:

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

---

<sup>1</sup>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 realtà nei sistemi Unix esiste un’altro modo per definire la funzione `main`, che prevede la presenza di un terzo parametro, `char *envp[]`, che fornisce l’ambiente (vedi sez. 2.3.4) del programma; questa forma però non è prevista dallo standard POSIX.1 per cui se si vogliono scrivere programmi portabili è meglio evitarla.

### 2.1.2 Come chiudere un programma

Normalmente un programma finisce quando la funzione `main` ritorna, una modalità equivalente di chiudere il programma è quella di chiamare direttamente la funzione `exit` (che viene comunque chiamata automaticamente quando `main` ritorna). Una forma alternativa è quella di chiamare direttamente la system call `_exit`, che restituisce il controllo direttamente alla routine di conclusione dei processi del kernel.

Oltre alla conclusione “normale” esiste anche la possibilità di una conclusione “anomala” del programma a causa della ricezione di un segnale (si veda cap. 9) o della chiamata alla funzione `abort`; torneremo su questo in sez. 3.2.4.

Il valore di ritorno della funzione `main`, o quello usato nelle chiamate ad `exit` e `_exit`, viene chiamato *stato di uscita* (o *exit status*) e passato al processo che aveva lanciato il programma (in genere la shell). In generale si usa questo valore per fornire informazioni sulla riuscita o il fallimento del programma; l’informazione è necessariamente generica, ed il valore deve essere compreso fra 0 e 255.

La convenzione in uso pressoché universale è quella di restituire 0 in caso di successo e 1 in caso di fallimento; l’unica eccezione 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. È opportuno adottare una di queste convenzioni a seconda dei casi. 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.

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 questa convenzione non sia universalmente seguita è una buona idea tenerne conto.

Si tenga presente inoltre che non è una buona idea usare il codice di errore restituito dalla variabile `errno` (per i dettagli si veda sez. 8.5) come stato di uscita. In generale infatti una shell non si cura del valore se non per vedere se è diverso da zero; inoltre il valore dello stato di uscita è sempre troncato ad 8 bit, per cui si potrebbe incorrere nel caso in cui restituendo un codice di errore 256, si otterebbe uno stato di uscita uguale a zero, che verrebbe interpretato come un successo.

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. In Linux esse sono poste rispettivamente ai valori di tipo `int` 0 e 1.

### 2.1.3 Le funzioni `exit` e `_exit`

Come accennato le funzioni usate per effettuare un’uscita “normale” da un programma sono due, la prima è la funzione `exit`, che è definita dallo standard ANSI C ed il cui prototipo è:

```
#include <stdlib.h>
void exit(int status)
    Causa la conclusione ordinaria del programma.

    La funzione non ritorna. Il processo viene terminato.
```

La funzione `exit` è pensata per eseguire una conclusione pulita di un programma che usa le librerie standard del C; essa esegue tutte le funzioni che sono state registrate con `atexit`

e `on_exit` (vedi sez. 2.1.4), e chiude tutti gli stream effettuando il salvataggio dei dati sospesi (chiamando `fclose`, vedi sez. 7.2.1), infine passa il controllo al kernel chiamando `_exit` e restituendo il valore di `status` come stato di uscita.

La system call `_exit` restituisce direttamente il controllo al kernel, concludendo immediatamente il processo; i dati sospesi nei buffer degli stream non vengono salvati e le eventuali funzioni registrate con `atexit` e `on_exit` non vengono eseguite. Il prototipo della funzione è:

```
#include <unistd.h>
void _exit(int status)
    Causa la conclusione immediata del programma.

La funzione non ritorna. Il processo viene terminato.
```

La funzione chiude tutti i file descriptor appartenenti al processo (si tenga presente che questo non comporta il salvataggio dei dati bufferizzati degli stream), fa sì che ogni figlio del processo sia ereditato da `init` (vedi cap. 3), manda un segnale `SIGCHLD` al processo padre (vedi sez. 9.2.6) ed infine ritorna lo stato di uscita specificato in `status` che può essere raccolto usando la funzione `wait` (vedi sez. 3.2.5).

#### 2.1.4 Le funzioni `atexit` e `on_exit`

Un'esigenza comune che si incontra nella programmazione è quella di dover effettuare una serie di operazioni di pulizia (ad esempio salvare dei dati, ripristinare delle impostazioni, eliminare dei file temporanei, ecc.) prima della conclusione di un programma. 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 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 (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>
void atexit(void (*function)(void))
    Registra la funzione function per la chiamata all'uscita dal programma.

La funzione restituisce 0 in caso di successo e -1 in caso di fallimento, errno non viene modificata.
```

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

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

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

La funzione restituisce 0 in caso di successo e -1 in caso di fallimento, errno non viene modificata.
```

In questo caso la funzione da chiamare all'uscita prende i due argomenti specificati nel prototipo, 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.

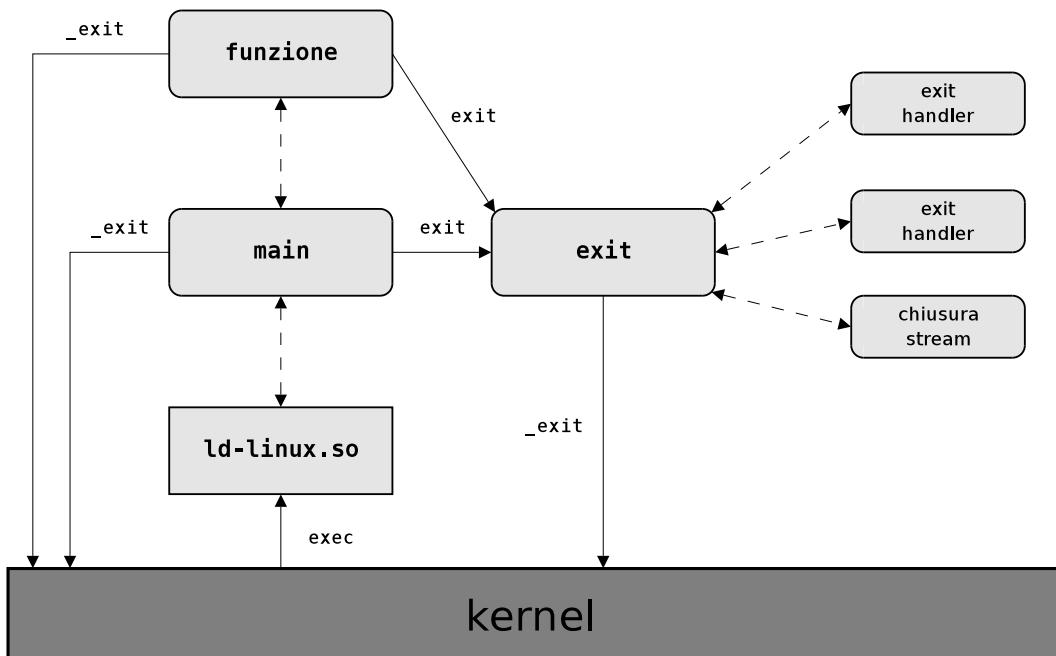
Nella sequenza di chiusura tutte le funzioni registrate verranno chiamate in ordine inverso rispetto a quello di registrazione (ed una stessa funzione registrata più volte sarà chiamata più volte); poi verranno chiusi tutti gli stream aperti, infine verrà chiamata `_exit`.

### 2.1.5 Conclusioni

Data l'importanza dell'argomento è opportuno sottolineare ancora una volta che in un sistema Unix l'unico modo in cui un programma può essere eseguito dal kernel è attraverso la chiamata alla system call `execve` (o attraverso una delle funzioni della famiglia `exec` che vedremo in sez. 3.2.7).

Allo stesso modo l'unico modo in cui un programma può concludere volontariamente la sua esecuzione è attraverso una chiamata alla system call `_exit`, o 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.

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); torneremo su questo aspetto in cap. 9.

## 2.2 I processi e l'uso della memoria

Una delle risorse base 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 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 vari sistemi 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.<sup>2</sup>

Come accennato in cap. 1 questo spazio di indirizzi è virtuale e non corrisponde all'effettiva posizione dei dati nella RAM del computer; in genere 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 virtuale viene divisa in pagine di dimensione fissa (che ad esempio sono di 4kb su macchine a 32 bit e 8kb sulle alpha, valori strettamente connessi all'hardware di gestione della memoria),<sup>3</sup> e ciascuna pagina della memoria virtuale è associata ad un supporto che può essere una pagina di memoria reale o ad un dispositivo di stoccaggio secondario (in genere lo spazio disco riservato alla swap, o i file che contengono il codice).

Lo stesso pezzo di memoria reale (o di spazio disco) 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 e quelle della memoria fisica della macchina viene gestita in maniera trasparente dall'hardware di gestione della memoria (la *Memory Management Unit* del processore). 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, 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*; l'hardware di 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, a tempi molto più lunghi, dovuti all'intervento del kernel.

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. 2.2.7).

### 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

---

<sup>2</sup>nel 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.

<sup>3</sup>con le versioni più recenti del kernel è possibile anche utilizzare pagine di dimensioni maggiori, per sistemi con grandi quantitativi di memoria in cui l'uso di pagine troppo piccole comporta una perdita di prestazioni.

genera quello che viene chiamato un *segmentation fault*. Se si tenta cioè di leggere o scrivere da un indirizzo per il quale non esiste un'associazione della pagina 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 anche da processi che eseguono altri programmi nel caso delle librerie). Viene marcato in sola lettura per evitare sovrascritture accidentali (o maliziose) che ne modifichino le istruzioni.

Viene allocato da `exec` all'avvio del programma e resta invariato per tutto il tempo dell'esecuzione.

2. Il segmento dei dati o *data segment*. Contiene 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`). Di norma è diviso in due parti.

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

```
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.

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

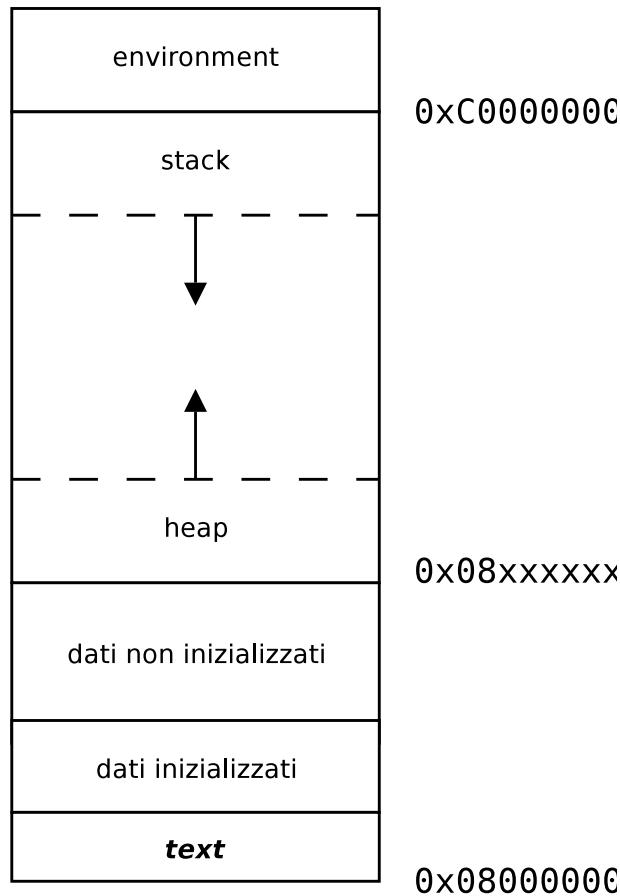
```
int vect[100];
```

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).<sup>4</sup>

Storicamente questo segmento viene chiamato BSS (da *block started by symbol*). La sua dimensione è fissa.

3. Lo *heap*. Tecnicamente lo si può considerare l'estensione del segmento dati, a cui di solito è posto giusto di seguito. È qui che avviene l'allocazione dinamica della memoria; può essere ridimensionato allocando e disallocando la memoria dinamica con le apposite funzioni (vedi sez. 2.2.3), ma il suo limite inferiore (quello adiacente al segmento dati) ha una posizione fissa.
4. Il segmento di *stack*, che contiene 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 (tipo il contenuto di alcuni registri della CPU). Poi la funzione chiamata alloca qui lo spazio per le sue variabili locali: in questo modo le funzioni possono essere chiamate ricorsivamente. Al ritorno della funzione lo spazio è automaticamente rilasciato e "ripulito". La pulizia in C e C++ viene fatta dal chiamante.<sup>5</sup>

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 di questi segmenti è riportata in fig. 2.2. Usando il comando `size` su un programma se ne può stampare le dimensioni dei segmenti di testo e di dati (inizializzati e BSS); si tenga presente però che il BSS non è mai salvato sul file che contiene l'eseguibile, dato che viene sempre inizializzato a zero al caricamento del programma.

### 2.2.3 Allocazione della memoria per i programmi C

Il C supporta, a livello di linguaggio, 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 (le cosiddette *variabili automatiche*), che esistono solo per la durata della funzione. 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

<sup>4</sup>si ricordi che questo vale solo per le variabili che vanno nel segmento dati, e non è affatto vero in generale.

<sup>5</sup>a meno che non sia stato specificato l'utilizzo di una calling convention diversa da quella standard.

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 le librerie del C forniscono una serie opportuna di funzioni per eseguire l'allocazione dinamica di memoria (in genere nello heap). Le variabili il cui contenuto è allocato in questo modo non potranno essere usate direttamente come le altre, ma l'accesso sarà possibile solo in maniera indiretta, attraverso dei puntatori.

#### 2.2.4 Le funzioni `malloc`, `calloc`, `realloc` e `free`

Le funzioni previste dallo standard ANSI C per la gestione della memoria sono quattro: `malloc`, `calloc`, `realloc` e `free`, i loro prototipi sono i seguenti:

```
#include <stdlib.h>
void *calloc(size_t size)
    Alloca size byte nello heap. La memoria viene inizializzata a 0.
    La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL
    in caso di fallimento, nel qual caso errno assumerà il valore ENOMEM.
void *malloc(size_t size)
    Alloca size byte nello heap. La memoria non viene inizializzata.
    La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL
    in caso di fallimento, nel qual caso errno assumerà il valore ENOMEM.
void *realloc(void *ptr, size_t size)
    Cambia la dimensione del blocco allocato all'indirizzo ptr portandola a size.
    La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e NULL
    in caso di fallimento, nel qual caso errno assumerà il valore ENOMEM.
void free(void *ptr)
    Disaloca lo spazio di memoria puntato da ptr.
    La funzione non ritorna nulla e non riporta errori.
```

Il puntatore ritornato dalle funzioni di allocazione è garantito essere sempre allineato correttamente per tutti i tipi di dati; ad esempio sulle macchine a 32 bit in genere è allineato a multipli di 4 byte e sulle macchine a 64 bit a multipli di 8 byte.

In genere si usano le funzioni `malloc` e `calloc` per allocare dinamicamente la quantità di memoria necessaria al programma indicata da `size`,<sup>6</sup> e siccome 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.

La memoria allocata dinamicamente deve essere esplicitamente rilasciata usando `free`<sup>7</sup> una volta che non sia più necessaria. Questa funzione vuole come parametro un puntatore restituito da una precedente chiamata a una qualunque delle funzioni di allocazione che non sia già stato liberato da un'altra chiamata a `free`, in caso contrario il comportamento della funzione è indefinito.

La funzione `realloc` si usa invece per cambiare (in genere aumentare) la dimensione di un'area di memoria precedentemente allocata, la funzione vuole in ingresso il puntatore restituito dalla precedente chiamata ad una `malloc` (se è passato un valore `NULL` allora la funzione si comporta come `malloc`)<sup>8</sup> 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

<sup>6</sup>queste funzioni presentano un comportamento diverso fra le *glibc* e le *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)`.

<sup>7</sup>le *glibc* provvedono anche una funzione `cfree` definita per compatibilità con SunOS, che è deprecata.

<sup>8</sup>questo 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.

rialloca altrove un blocco della dimensione voluta, copiandoci automaticamente il contenuto; lo spazio aggiunto non viene inizializzato.

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.

Un errore abbastanza frequente (specie se si ha a che fare con vettori di puntatori) è quello di chiamare `free` più di una volta sullo stesso puntatore; per evitare questo problema una soluzione di ripiego è quella di assegnare sempre a `NULL` ogni puntatore liberato con `free`, dato che, quando il parametro è un puntatore nullo, `free` non esegue nessuna operazione.

Le *glibc* hanno un'implementazione delle routine di allocazione che è controllabile dall'utente attraverso alcune variabili di ambiente, in particolare diventa possibile tracciare questo tipo di errori usando la variabile di ambiente `MALLOC_CHECK_` che quando viene definita mette in uso una versione meno efficiente delle funzioni suddette, che però è più tollerante nei confronti di piccoli errori come quello di chiamate doppie a `free`. In particolare:

- se la variabile è posta a zero gli errori vengono ignorati.
- se è posta ad 1 viene stampato un avviso sullo *standard error* (vedi sez. 7.1.3).
- se è posta a 2 viene chiamata `abort`, che in genere causa l'immediata conclusione del programma.

Il problema più comune e più difficile da risolvere che si incontra con le routine 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 subroutine 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 subroutine continueranno ad effettuare altre allocazioni, 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 subroutine 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* è notevolmente ridimensionato attraverso l'uso accurato di appositi oggetti come gli *smartpointers*. Questo però va a scapito delle prestazioni dell'applicazione in esecuzione.

In altri linguaggi come il java e recentemente il C# il problema non si pone nemmeno perché la gestione della memoria viene fatta totalmente in maniera automatica, ovvero il programmatore non deve minimamente preoccuparsi di liberare la memoria allocata precedentemente quando non serve più, poiché il framework gestisce automaticamente la cosiddetta *garbage collection*. In tal caso, attraverso meccanismi simili a quelli del *reference counting*, quando una zona di memoria precedentemente allocata non è più riferita da nessuna parte del codice in esecuzione, può essere deallocata automaticamente in qualunque momento dall'infrastruttura.

Anche questo va a scapito delle prestazioni dell'applicazione in esecuzione (inoltre le applicazioni sviluppate con tali linguaggi di solito non sono eseguibili compilati, come avviene invece per il C ed il C++, ed è necessaria la presenza di una infrastruttura per la loro interpretazione e pertanto hanno di per sé delle prestazioni più scadenti rispetto alle stesse applicazioni compilate

direttamente). Questo comporta però il problema della non predicitività del momento in cui viene deallocata la memoria precedentemente allocata da un oggetto.

Per limitare l'impatto di questi problemi, e semplificare la ricerca di eventuali errori, l'implementazione delle routine di allocazione delle *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 routine di allocazione in grado, senza neanche ricompilare il programma,<sup>9</sup> di eseguire diagnostiche anche molto complesse riguardo l'allocazione della memoria.

### 2.2.5 La funzione `alloca`

Una possibile alternativa all'uso di `malloc`, che non soffre dei problemi di *memory leak* descritti in precedenza, è la funzione `alloca`, che invece di allocare la memoria nello heap usa il segmento di stack della funzione corrente. La sintassi è identica a quella di `malloc`, il suo prototipo è:

```
#include <stdlib.h>
void *alloca(size_t size)
    Alloca size byte nello stack.
```

La funzione restituisce il puntatore alla zona di memoria allocata in caso di successo e `NULL` in caso di fallimento, nel qual caso `errno` assumerà il valore `ENOMEM`.

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.

Come è evidente questa funzione ha molti vantaggi, 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. 2.4.4).

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. 2.4.3.

### 2.2.6 Le funzioni `brk` e `sbrk`

Queste due funzioni vengono utilizzate soltanto quando è necessario effettuare direttamente la gestione della memoria associata allo spazio dati di un processo, ad esempio qualora si debba implementare la propria versione delle routine di allocazione della memoria viste in sez. 2.2.4. La prima funzione è `brk`, ed il suo prototipo è:

---

<sup>9</sup>esempi sono *Dmalloc* <http://dmalloc.com/> di Gray Watson ed *Electric Fence* di Bruce Perens.

```
#include <unistd.h>
int brk(void *end_data_segment)
    Sposta la fine del segmento dati.
```

La funzione restituisce 0 in caso di successo e -1 in caso di fallimento, nel qual caso `errno` assumerà il valore `ENOMEM`.

La funzione è un’interfaccia diretta all’omonima system call ed imposta l’indirizzo finale del segmento dati di un processo all’indirizzo specificato da `end_data_segment`. Quest’ultimo deve essere un valore ragionevole, ed inoltre la dimensione totale del segmento non deve comunque eccedere un eventuale limite (si veda sez. 8.3.2) imposto sulle dimensioni massime dello spazio dati del processo.

La seconda funzione per la manipolazione delle dimensioni del segmento dati<sup>10</sup> è `sbrk`, ed il suo prototipo è:

```
#include <unistd.h>
void *sbrk(ptrdiff_t increment)
    Incrementa la dimensione dello spazio dati.
```

La funzione restituisce il puntatore all’inizio della nuova zona di memoria allocata in caso di successo e `NULL` in caso di fallimento, nel qual caso `errno` assumerà il valore `ENOMEM`.

la funzione incrementa la dimensione lo spazio dati di un programma di `increment` byte, restituendo il nuovo indirizzo finale dello stesso. Un valore nullo permette di ottenere l’attuale posizione della fine del segmento dati.

Queste funzioni sono state deliberatamente escluse dallo standard POSIX.1 e per i programmi normali è sempre opportuno usare le funzioni di allocazione standard descritte in precedenza, che sono costruite su di esse.

### 2.2.7 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 nello 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. 3.4.3).

- *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 può cancellare la memoria su cui scrive le sue variabili, ma non può toccare

<sup>10</sup>in questo caso si tratta soltanto di una funzione di libreria, e non di una system call.

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.

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*. Infine *memory lock* non sono ereditati dai processi figli.<sup>11</sup>

Siccome la richiesta di un *memory lock* da parte di un processo riduce la memoria fisica disponibile nel sistema, questo ha un evidente impatto su tutti gli altri processi, per cui solo un processo con i privilegi di amministratore (vedremo in sez. 3.3 cosa significa) ha la capacità di bloccare una pagina. Ogni processo può però sbloccare le pagine relative alla propria memoria.

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* e la costante `PAGESIZE` in `limits.h` per indicare la dimensione di una pagina in byte.

Le funzioni per bloccare e sbloccare la paginazione di singole sezioni di memoria sono `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 seguenti:

<b>ENOMEM</b>	alcuni indirizzi dell'intervallo specificato non corrispondono allo spazio di indirizzi del processo o si è ecceduto il numero massimo consentito di pagine bloccate.
<b>EINVAL</b>	<code>len</code> non è un valore positivo.

e, per `mlock`, anche `EPERM` quando il processo non ha i privilegi richiesti per l'operazione.

Le due funzioni permettono rispettivamente di bloccare e sbloccare la paginazione per l'intervallo di memoria specificato dagli argomenti, che ne indicano nell'ordine l'indirizzo iniziale e la lunghezza. Tutte le pagine che contengono una parte dell'intervallo bloccato sono mantenute in RAM per tutta la durata del blocco.

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

```
#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`.

<sup>11</sup>ma siccome Linux usa il *copy on write* (vedi sez. 3.2.2) gli indirizzi virtuali del figlio sono mantenuti sullo stesso segmento di RAM del padre, quindi fintanto che un figlio non scrive su un segmento, può usufruire del *memory lock* del padre.

L'argomento `flags` di `mlockall` permette di controllarne il comportamento; esso può essere specificato come l'OR aritmetico delle due costanti:

`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.

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 dei flag permette di selezionare con maggior finezza le pagine da bloccare, ad esempio limitandosi a tutte le pagine allocate a partire da un certo momento.

In ogni caso un processo real-time che deve entrare in una sezione critica deve provvedere a riservare memoria sufficiente prima dell'ingresso, per scongiurare l'occorrenza di un eventuale *page fault* causato dal meccanismo di *copy on write*. Infatti se nella sezione critica si va ad utilizzare memoria che non è ancora stata riportata in RAM si potrebbe avere un *page fault* durante l'esecuzione della stessa, con conseguente rallentamento (probabilmente inaccettabile) dei tempi di esecuzione.

In genere si ovvia a questa problematica chiamando una funzione che ha allocato una quantità sufficientemente ampia di variabili automatiche, in modo che esse vengano mappate in RAM dallo stack, dopo di che, per essere sicuri che esse siano state effettivamente portate in memoria, ci si scrive sopra.

## 2.3 Argomenti, opzioni ed ambiente di un processo

Tutti i programmi hanno la possibilità di ricevere argomenti e opzioni quando vengono lanciati. Il passaggio degli argomenti è effettuato attraverso gli argomenti `argc` e `argv` della funzione `main`, che vengono passati al programma dalla shell (o dal processo che esegue la `exec`, secondo le modalità che vedremo in sez. 3.2.7) quando questo viene messo in esecuzione.

Oltre al passaggio degli argomenti, un'altra modalità che permette di passare delle informazioni che modifichino il comportamento di un programma è quello dell'uso del cosiddetto *environment* (cioè l'uso delle *variabili di ambiente*). In questa sezione esamineremo le funzioni che permettono di gestire argomenti ed opzioni, e quelle che consentono di manipolare ed utilizzare le variabili di ambiente.

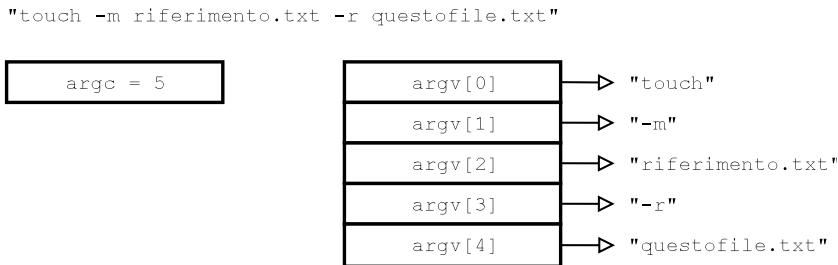
### 2.3.1 Il formato degli argomenti

In genere passaggio degli argomenti al programma viene effettuato dalla shell, che si incarica di leggere la linea di comando e di effettuarne la scansione (il cosiddetto *parsing*) per individuare le parole che la compongono, ciascuna delle quali viene considerata un argomento. Di norma per individuare le parole viene usato come carattere di separazione lo spazio o il tabulatore, ma il comportamento è modificabile attraverso l'impostazione della variabile di ambiente `IFS`.

Nella scansione viene costruito il vettore di puntatori `argv` inserendo in successione il puntatore alla stringa costituente l'*n*-simo parametro; la variabile `argc` viene inizializzata al numero di argomenti trovati, in questo modo il primo parametro è sempre il nome del programma; un esempio di questo meccanismo è mostrato in fig. 2.3.

### 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` che inizia con



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

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 comando 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 dei argomenti a linea di comando passati in `argv` le librerie standard del C forniscono la funzione `getopt`, che ha il seguente prototipo:

```
#include <unistd.h>
int getopt(int argc, char *const argv[], const char *optstring)
    Esegue il parsing degli argomenti passati da linea di comando riconoscendo le possibili
    opzioni segnalate con optstring.

    Ritorna il carattere che segue l’opzione, ':' se manca un parametro all’opzione, '?' se l’opzione
    è sconosciuta, e -1 se non esistono altre opzioni.
```

Questa funzione prende come argomenti le due variabili `argc` e `argv` passate a `main` ed una stringa che indica quali sono le opzioni valide; la funzione effettua la scansione della lista degli argomenti ricercando ogni stringa che comincia con `-` 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 un segno 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 la funzione trova un’opzione essa ritorna il valore numerico del carattere, in questo modo si possono eseguire azioni specifiche usando uno `switch`; `getopt` 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.

In fig. 2.4 è mostrata la sezione del programma `ForkTest.c` (che useremo nel prossimo capitolo per effettuare dei test sulla creazione dei processi) deputata alla decodifica delle opzioni a riga di comando.

---

```

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.

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 Opzioni in formato esteso

Un'estensione di questo schema è costituito dalle cosiddette *long-options* espresse nella forma `-option=parameter`, anche la gestione di queste ultime è stata standardizzata attraverso l'uso di una versione estesa di `getopt`.

(NdA: da finire).

### 2.3.4 Le variabili di ambiente

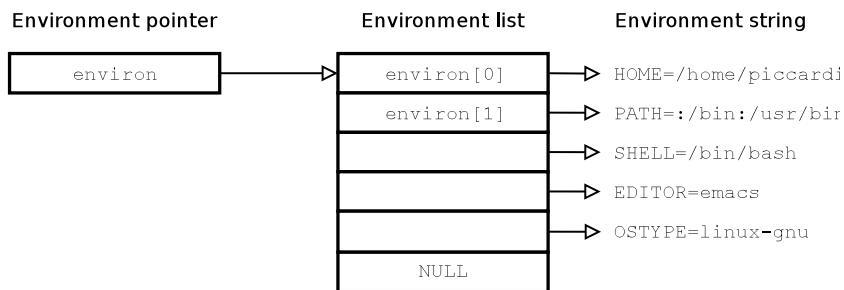
Oltre agli argomenti passati a linea di comando ogni processo riceve dal sistema un *ambiente*, nella forma di una lista di variabili (detta *environment list*) messa a disposizione dal processo, e costruita nella chiamata alla funzione `exec` quando questo viene lanciato.

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`, 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*. 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.<sup>12</sup>

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. 3.2.7).

La shell ad esempio ne usa molte per il suo funzionamento (come `PATH` per la ricerca dei comandi, o `IFS` per la scansione degli argomenti), e alcune di esse (come `HOME`, `USER`, etc.) sono definite al login (per i dettagli si veda sez. 10.1.4). In genere è cura dell'amministratore 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).

Gli standard POSIX e XPG3 definiscono alcune di queste variabili (le più comuni), come riportato in tab. 2.1. GNU/Linux le supporta tutte e ne definisce anche altre: per una lista più completa si può controllare `man environ`.

<sup>12</sup>la convenzione vuole che si usino dei nomi maiuscoli per le variabili di ambiente di uso generico, i nomi minuscoli sono in genere riservati alle variabili interne degli script di shell.

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

**Tabella 2.1:** 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:

```
#include <stdlib.h>
char *getenv(const char *name)
    Esamina l'ambiente del processo cercando una stringa che corrisponda a quella specificata
    da name.

La funzione ritorna NULL se non trova nulla, o il puntatore alla stringa che corrisponde (di solito
nella forma NOME=valore).
```

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 e per cancellare le variabili di ambiente. Uno schema delle funzioni previste nei vari standard e disponibili in Linux è riportato in tab. 2.2.

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

**Tabella 2.2:** Funzioni per la gestione delle variabili di ambiente.

In Linux<sup>13</sup> sono definite tutte le funzioni elencate in tab. 2.2. La prima, `getenv`, l'abbiamo appena esaminata; delle restanti le prime due, `putenv` e `setenv`, servono per assegnare nuove variabili di ambiente, i loro prototipi sono i seguenti:

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite)
    Imposta la variabile di ambiente name al valore value.
int putenv(char *string)
    Aggiunge la stringa string all'ambiente.

Entrambe le funzioni ritornano 0 in caso di successo e -1 per un errore, che è sempre ENOMEM.
```

la terza, `unsetenv`, serve a cancellare una variabile di ambiente; il suo prototipo è:

```
#include <stdlib.h>
void unsetenv(const char *name)
    Rimuove la variabile di ambiente name.
```

<sup>13</sup>in realtà nelle libc4 e libc5 sono definite solo le prime quattro, `clearenv` è stata introdotta con le glibc 2.0.

questa funzione elimina ogni occorrenza della variabile specificata; se essa non esiste non succede nulla. Non è prevista (dato che la funzione è `void`) nessuna segnalazione di errore.

Per modificare o aggiungere una variabile di ambiente si possono usare sia `setenv` che `putenv`. La prima permette di specificare separatamente nome e valore della variabile di ambiente, inoltre il valore di `overwrite` specifica il comportamento della funzione nel caso la variabile esista già, sovrascrivendola se diverso da zero, lasciandola immutata se uguale a zero.

La seconda funzione prende come parametro una stringa analoga quella restituita da `getenv`, e sempre nella forma `NOME=valore`. Se la variabile specificata non esiste la stringa sarà aggiunta all’ambiente, se invece esiste il suo valore sarà impostato a quello specificato da `string`. Si tenga presente che, seguendo lo standard SUSv2, le *glibc* successive alla versione 2.1.2 aggiungono<sup>14</sup> `string` alla lista delle variabili di ambiente; 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. 2.4.3).

Si tenga infine presente che se si passa a `putenv` solo il nome di una variabile (cioè `string` è nella forma `NAME` e non contiene un carattere ‘=’) allora questa viene cancellata dall’ambiente. Infine se la chiamata di `putenv` comporta la necessità di allocare una nuova versione del vettore `environ` questo sarà allocato, ma la versione corrente sarà deallocated solo se anch’essa è risultante da un’allocazione fatta in precedenza da un’altra `putenv`. Questo perché il vettore delle variabili di ambiente iniziale, creato dalla chiamata ad `exec` (vedi sez. 3.2.7) è piazzato al di sopra dello stack, (vedi fig. 2.2) e non nello heap e non può essere deallocated. Inoltre la memoria associata alle variabili di ambiente eliminate non viene liberata.

L’ultima funzione è `clearenv`, 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 restituisce 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. In tal caso si provvede alla cancellazione di tutto l’ambiente per costruirne una versione “sicura” da zero.

## 2.4 Problematiche di programmazione generica

Benché questo non sia un libro di C, è opportuno affrontare alcune delle problematiche generali che possono emergere nella programmazione 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.

### 2.4.1 Il passaggio delle variabili e dei valori di ritorno

Una delle caratteristiche standard del C è che le variabili vengono passate alle subroutine 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 `&`).

<sup>14</sup>il comportamento è lo stesso delle vecchie *libc4* e *libc5*; nelle *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 dalle 2.1.2, eliminando anche, sempre in conformità a SUSv2, l’attributo `const` dal prototipo.

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

Questo però va inteso nella maniera corretta. Il passaggio *by value* vale per qualunque variabile, puntatori compresi; quando però in una subroutine 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 sono sempre gli stessi, e le eventuali modifiche avranno effetto e saranno visibili anche nella routine chiamante.

Nella maggior parte delle funzioni di libreria e delle system call i puntatori vengono usati per scambiare dati (attraverso buffer o strutture) e le variabili semplici vengono usate per specificare argomenti; in genere le informazioni a riguardo dei risultati vengono passate alla routine 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. Per far questo si usa il cosiddetto *value result argument*, si passa cioè, invece di una normale variabile, un puntatore alla stessa; vedremo alcuni esempi di questa modalità 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 questo meccanismo.

## 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 della *ellipsis* `...` nella dichiarazione della funzione; ma non provvede a livello di linguaggio alcun meccanismo con cui dette funzioni possono accedere ai loro argomenti.

L'accesso viene invece realizzato dalle librerie standard che provvedono gli strumenti adeguati. L'uso delle *variadic function* prevede tre punti:

- *Dichiarare* la funzione come *variadic* usando un prototipo che contenga una *ellipsis*.
- *Definire* la funzione come *variadic* usando lo stesso *ellipsis*, ed utilizzare le apposite macro che consentono la gestione di un numero variabile di argomenti.
- *Chiamare* la funzione specificando prima gli argomenti fissi, e a seguire gli 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 `exec1` che vedremo in sez. 3.2.7:

```
int exec1(const char *path, const char *arg, ...);
```

in questo caso la funzione prende due argomenti fissi ed un numero variabile di altri argomenti (che verranno 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*<sup>15</sup> 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 parametro fisso come `register`.

---

<sup>15</sup>il 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.

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; essi verranno estratti dallo stack secondo l'ordine in cui sono stati scritti. Per fare questo in `stdarg.h` sono definite delle apposite macro; la procedura da seguire è la seguente:

1. Inizializzare un puntatore alla lista degli argomenti di tipo `va_list` attraverso la macro `va_start`.
2. Accedere ai vari argomenti opzionali 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`.

In generale è perfettamente legittimo richiedere meno argomenti di quelli che potrebbero essere stati effettivamente forniti, e 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 forniti si otterranno dei valori indefiniti. Nel caso del `gcc` l'uso della macro `va_end` è inutile, ma si consiglia di usarlo ugualmente per compatibilità.

Le definizioni delle tre macro sono le seguenti:

```
#include <stdarg.h>
void va_start(va_list ap, last)
    Inizializza il puntatore alla lista di argomenti ap; il parametro last deve essere l'ultimo
    degli argomenti fissi.
type va_arg(va_list ap, type)
    Restituisce il valore del successivo parametro opzionale, modificando opportunamente ap;
    la macro richiede che si specifichi il tipo dell'argomento attraverso il parametro type che
    deve essere il nome del tipo dell'argomento in questione. Il tipo deve essere self-promoting.
void va_end(va_list ap)
    Conclude l'uso di ap.
```

In generale si possono avere più puntatori alla lista degli argomenti, ciascuno andrà inizializzato con `va_start` e letto con `va_arg` e ciascuno potrà scandire la lista degli argomenti per conto suo.

Dopo l'uso di `va_end` la variabile `ap` diventa indefinita e successive chiamate a `va_arg` non funzioneranno. Si avranno risultati indefiniti anche chiamando `va_arg` specificando un tipo che non corrisponde a quello del parametro.

Un altro limite delle 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 una subroutine passandole il puntatore alla lista di argomenti; in questo caso però si richiede che al ritorno della funzione il puntatore non venga più usato (lo standard richiederebbe la chiamata esplicita di `va_end`), 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. La cosa più naturale in questo caso sembrerebbe quella di copiarsi il puntatore alla lista degli argomenti con una semplice assegnazione. 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, per questo motivo `va_list` è definito come *tipo opaco* e non può essere assegnato direttamente ad un'altra variabile dello stesso tipo. Per risolvere questo problema lo standard ISO C99<sup>16</sup> ha previsto una macro ulteriore che permette di eseguire la copia di un puntatore alla lista degli argomenti:

<sup>16</sup>alcuni sistemi che non hanno questa macro provvedono al suo posto `__va_copy` che era il nome proposto in una bozza dello standard.

```
#include <stdarg.h>
void va_copy(va_list dest, va_list src)
    Copia l'attuale valore src del puntatore alla lista degli argomenti su 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`).

Uno 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 passati effettivamente 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 parametro per specificare anche il tipo degli argomenti (come fa la stringa di formato per `printf`).

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

### 2.4.3 Potenziali problemi con le variabili automatiche

Uno dei possibili problemi che si possono avere con le subroutine è quello di restituire alla funzione chiamante dei dati che sono contenuti in una variabile automatica. Ovviamente quando la subroutine ritorna la sezione dello stack che conteneva la variabile automatica potrà essere riutilizzata da una nuova funzione, con le immaginabili conseguenze 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 variabili locali; qualora sia necessario utilizzare variabili che possano essere viste anche dalla funzione chiamante queste devono essere allocate esplicitamente, o in maniera statica (usando variabili di tipo `static` o `extern`), o dinamicamente con una delle funzioni della famiglia `malloc`.

### 2.4.4 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 da [1] che da [3], 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 si ottengono le indicazioni per l'esecuzione delle opportune operazioni.

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.<sup>17</sup>

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 nella funzione da cui si era partiti, quando serve. 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 zero quando è chiamata direttamente e 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`<sup>18</sup> 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 routine 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 al ritorno della `setjmp` con cui si era salvato `env`, che restituirà il valore `val` invece di zero. Il valore di `val` specificato nella chiamata deve essere diverso da zero, se si è specificato 0 sarà comunque restituito 1 al suo posto.

In sostanza un `longjmp` è analogo ad un `return`, solo che invece di ritornare alla riga successiva della funzione chiamante, il programma ritorna alla posizione della relativa `setjmp`, l’altra differenza 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 particolare `setjmp` è implementata con una macro, pertanto non si può cercare di ottenerne l’indirizzo, ed inoltre delle chiamate a questa funzione sono sicure solo in uno dei seguenti casi:

<sup>17</sup>a meno che, come precisa [3], 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.

<sup>18</sup>questo è un classico esempio di variabile di *tipo opaco*. Si definiscono così strutture ed altri oggetti usati da una libreria, la cui struttura interna non deve essere vista dal programma chiamante (da cui il nome) che li devono utilizzare solo attraverso delle opportune funzioni di gestione.

- 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 la chiamata diretta e quella ottenuta da un `longjmp`, è il valore di ritorno di `setjmp`, essa è usualmente chiamata all'interno di un comando `if`.

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`<sup>19</sup>) 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`<sup>20</sup>.

---

<sup>19</sup>la 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 dato che tutti i compilatori sono normalmente in grado di valutare con maggior efficacia degli stessi programmatore quando sia il caso di eseguire questa ottimizzazione.

<sup>20</sup>la 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).



# Capitolo 3

## La gestione dei processi

Come accennato nell'introduzione in un sistema Unix 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 Introduzione

Inizieremo con un'introduzione generale ai concetti che stanno alla base della gestione dei processi in un sistema unix-like. Introdurremo in questa sezione l'architettura della gestione dei processi e le sue principali caratteristiche, dando una panoramica sull'uso delle principali funzioni di gestione.

#### 3.1.1 L'architettura della gestione dei processi

A differenza di quanto avviene in altri sistemi (ad esempio nel VMS la generazione di nuovi processi è un'operazione privilegiata) una delle caratteristiche di Unix (che esamineremo in dettaglio più avanti) è che qualunque processo può a sua volta generarne altri, detti processi figli (*child process*). Ogni processo è identificato presso il sistema da un numero univoco, il cosiddetto *process identifier* o, più brevemente, *pid*, assegnato in forma progressiva (vedi sez. 3.2.1) quando il processo viene creato.

Una seconda caratteristica di un sistema Unix 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 è che ogni processo è sempre stato generato da un altro, che 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 speciale (che normalmente è `/sbin/init`), che viene lanciato dal kernel alla conclusione della fase di avvio; essendo questo il primo processo lanciato dal sistema ha sempre il *pid* uguale a 1 e non è figlio di nessun altro processo.

Ovviamente `init` è un processo speciale che in genere si occupa di far partire 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. 3.2.4) 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, passando la riga `init=/bin/sh` come parametro di avvio.

```
[piccardi@gont piccardi]$ pstree -n
init--keventd
|-kapm-idled
|-kreiserfsd
|-portmap
|-syslogd
|-klogd
|-named
|-rpc.statd
|-gpm
|-inetd
|-junkbuster
|-master--qmgr
|   '-pickup
|-sshd
|-xfs
|-cron
|-bash---startx---xinit---XFree86
|   '-WindowMaker---ssh-agent
|       |-wmtime
|       |-wmmon
|       |-wmmount
|       |-wmppp
|       |-wmcube
|       |-wmmixer
|       |-wmgttemp
|       |-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`.

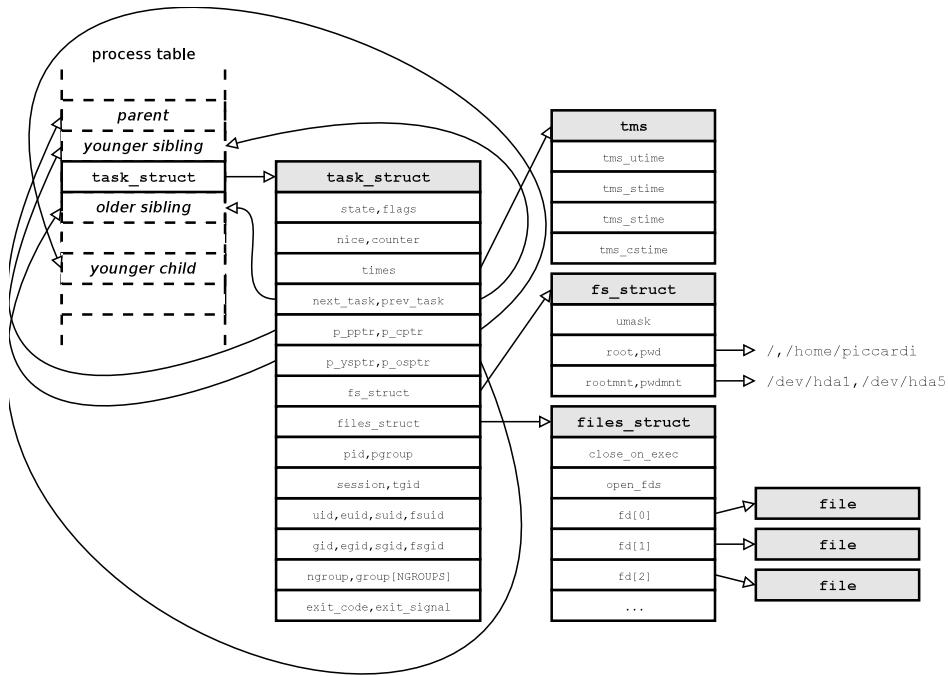
Dato che tutti i processi attivi nel sistema sono comunque generati da `init` o da uno dei suoi figli<sup>1</sup> si possono classificare i processi con la relazione padre/figlio in un'organizzazione gerarchica ad albero, in maniera analoga a come i file sono organizzati in un albero di directory (si veda sez. 4.1.1); 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.

Il kernel mantiene una tabella dei processi attivi, la cosiddetta *process table*; per ciascun processo viene mantenuta una voce, costituita da una struttura `task_struct`, nella tabella dei processi che contiene tutte le informazioni rilevanti per quel processo. Tutte le strutture usate a questo scopo sono dichiarate nell'header file `linux/sched.h`, ed uno schema semplificato, che riporta la struttura delle principali informazioni contenute nella `task_struct` (che in seguito incontreremo a più riprese), è mostrato in fig. 3.2.

Come accennato in sez. 1.1 è lo *scheduler* che decide quale processo mettere in esecuzione;

---

<sup>1</sup>in realtà questo non è del tutto vero, in Linux ci sono alcuni processi speciali che pur comparendo come figli di `init`, o con `pid` successivi, sono in realtà generati direttamente dal kernel, (come `keventd`, `kswapd`, etc.).



**Figura 3.2:** Schema semplificato dell'architettura delle strutture usate dal kernel nella gestione dei processi.

esso viene eseguito ad ogni system call ed ad ogni interrupt,<sup>2</sup> (ma può essere anche attivato esplicitamente). Il timer di sistema provvede comunque a che esso sia invocato periodicamente, generando un interrupt periodico secondo la frequenza specificata dalla costante HZ, definita in `asm/param.h`, ed il cui valore è espresso in Hertz.<sup>3</sup>

Ogni volta che viene eseguito, lo *scheduler* effettua il calcolo delle priorità dei vari processi attivi (torneremo su questo in sez. 3.4) e stabilisce quale di essi debba essere posto in esecuzione fino alla successiva invocazione.

### 3.1.2 Una panoramica sulle funzioni fondamentali

In un sistema unix-like i processi vengono sempre creati da altri processi tramite la funzione `fork`; il nuovo processo (che viene chiamato *figlio*) creato dalla `fork` è una copia identica del processo processore originale (detto *padre*), ma ha un nuovo *pid* e viene eseguito in maniera indipendente (le differenze fra padre e figlio sono affrontate in dettaglio in sez. 3.2.2).

Se si vuole che il processo padre si fermi fino alla conclusione del processo figlio questo deve essere specificato subito dopo la `fork` chiamando la funzione `wait` o la funzione `waitpid` (si veda sez. 3.2.5); queste funzioni restituiscono anche un'informazione abbastanza limitata sulle cause della terminazione del processo figlio.

Quando un processo ha concluso il suo compito o ha incontrato un errore non risolvibile esso può essere terminato con la funzione `exit` (si veda quanto discusso in sez. 2.1.2). La vita del processo però termina solo quando la notifica della sua conclusione viene ricevuta dal processo padre, a quel punto tutte le risorse allocate nel sistema ad esso associate vengono rilasciate.

Avere due processi che eseguono esattamente lo stesso codice non è molto utile, normalmente si genera un secondo processo per affidargli l'esecuzione di un compito specifico (ad esempio gestire una connessione dopo che questa è stata stabilita), o fargli eseguire (come fa la shell) un altro

<sup>2</sup>più in una serie di altre occasioni. NDT completare questa parte.

<sup>3</sup>Il valore usuale di questa costante è 100, per tutte le architetture eccetto l'alpha, per la quale è 1000. Occorre fare attenzione a non confondere questo valore con quello dei clock tick (vedi sez. 8.4.1).

programma. Per quest'ultimo caso si usa la seconda funzione fondamentale per programmazione coi processi che è la `exec`.

Il programma che un processo sta eseguendo si chiama immagine del processo (o *process image*), le funzioni della famiglia `exec` permettono di caricare un altro programma da disco sostituendo quest'ultimo all'immagine corrente; questo fa sì che l'immagine precedente venga completamente cancellata. Questo significa che quando il nuovo programma termina, anche il processo termina, e non si può tornare alla precedente immagine.

Per questo motivo la `fork` e la `exec` sono funzioni molto particolari con caratteristiche uniche rispetto a tutte le altre, infatti la prima ritorna due volte (nel processo padre e nel figlio) mentre la seconda non ritorna mai (in quanto con essa viene eseguito un altro programma).

## 3.2 Le funzioni di base

In questa sezione tratteremo le problematiche della gestione dei processi all'interno del sistema, illustrandone tutti i dettagli. Inizieremo con 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.

### 3.2.1 Gli identificatori dei processi

Come accennato nell'introduzione, ogni processo viene identificato dal sistema da un numero identificativo univoco, il *process ID* o *pid*; quest'ultimo è un tipo di dato standard, il `pid_t` che in genere è un intero con segno (nel caso di Linux e delle *glibc* il tipo usato è `int`).

Il *pid* viene assegnato in forma progressiva<sup>4</sup> ogni volta che un nuovo processo viene creato, fino ad un limite che, essendo 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,<sup>5</sup> 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 `getpid` e `getppid`, i cui prototipi sono:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void)
    Restituisce il pid del processo corrente.
pid_t getppid(void)
    Restituisce il pid 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 `ForkTest.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 `tmpname` (si veda sez. 5.1.8) usa il *pid* per generare un *pathname* univoco, che non potrà essere replicato da un altro processo che usi la stessa funzione.

<sup>4</sup>in 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. 10.1.2).

<sup>5</sup>questi valori, fino al kernel 2.4.x, sono definiti dalla macro `PID_MAX` in `threads.h` e direttamente in `fork.c`, con il kernel 2.5.x e la nuova interfaccia per i thread creata da Ingo Molnar anche il meccanismo di allocazione dei *pid* è stato modificato.

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. 10, 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. 10.1.2, relativi al controllo di sessione), ad ogni processo vengono associati degli altri identificatori 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. 3.3.

### 3.2.2 La funzione fork

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

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

In caso di successo restituisce il *pid* del figlio al padre e zero al figlio; ritorna -1 al padre (senza creare il figlio) in caso di errore; *errno* può assumere i valori:

<b>EAGAIN</b>	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.
<b>ENOMEM</b>	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 però una copia del padre, e riceve una copia dei segmenti di testo, stack e dati (vedi sez. 2.2.2), ed esegue esattamente lo stesso codice del padre. Si tenga presente però che la memoria è copiata, non condivisa, pertanto padre e figlio vedono variabili diverse.

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 read-only 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 modo che gli permette di identificare quello appena creato; al contrario un figlio ha sempre un solo padre (il cui *pid* può sempre essere ottenuto con **getppid**, vedi sez. 3.2.1) per cui si usa il 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

---

```

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.

si è ecceduto il limite sul numero totale di processi permessi all'utente (vedi sez. 8.3.2, ed in particolare tab. 8.12).

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 il 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. 3.2.7) 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 gli attributi del processo (maschera dei segnali, redirezione dell'output, identificatori) 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 gestisce le opzioni a riga di comando, è disponibile nel file `ForkTest.c`, distribuito insieme agli altri sorgenti degli esempi su [http://gapil.firenze.linux.it/gapil\\_source.tgz](http://gapil.firenze.linux.it/gapil_source.tgz).

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<sup>6</sup> senza specificare attese (come si può notare in (17-19) i valori predefiniti specificano di non attendere), otterremo come output 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<sup>7</sup> 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,

---

<sup>6</sup>che è preceduto dall'istruzione `export LD_LIBRARY_PATH=.` per permettere l'uso delle librerie dinamiche.

<sup>7</sup>a partire dal kernel 2.5.2-pre10 è stato introdotto il nuovo scheduler di Ingo Molnar che esegue sempre per primo il figlio; per mantenere la portabilità è opportuno non fare comunque affidamento su questo comportamento.

risultando del tutto impredicibile. 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 esplicati meccanismi di sincronizzazione, pena il rischio di incorrere nelle cosiddette *race condition* (vedi sez. 3.5.2).

Si noti inoltre che essendo i segmenti di memoria utilizzati dai singoli processi completamente separati, 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; per illustrarlo meglio proviamo a redirigere su un file l'output del nostro 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 cap. 6 e in cap. 7. Qui basta accennare che si sono usate le funzioni standard della libreria del C che prevedono l'output bufferizzato; e questa bufferizzazione (trattata in dettaglio in sez. 7.1.4) 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 (nel qual caso il buffer viene scaricato ad ogni carattere di a capo).

Nel primo esempio allora avevamo che ad ogni chiamata a `printf` il buffer veniva scaricato, e le singole righe erano stampate 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 della condivisione dei file è trattato in dettaglio in sez. 6.3.1), ma anche che, a differenza di quanto avviene per le variabili, la posizione corrente sul file è condivisa fra il padre e tutti i processi figli.

Quello che succede è che quando lo standard output del padre viene rediretto, lo stesso avviene anche per tutti i figli; la funzione `fork` infatti ha la caratteristica di duplicare nei figli tutti i file descriptor aperti nel padre (allo stesso modo in cui lo fa la funzione `dup`, trattata in sez. 6.3.4), il che comporta che padre e figli condividono le stesse voci della *file table* (per la spiegazione di questi termini si veda sez. 6.3.1) fra cui c'è anche la posizione corrente nel file.

In questo modo se un processo scrive sul file aggiornerà la posizione corrente 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 processi scrivono sullo stesso file, che l'output successivo di un processo vada a sovrapporsi a quello dei 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 è la shell quando lancia un programma, il cui output va sullo standard output).

In questo modo, anche se l'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 impredicibile. 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. 3.2.7 e sez. 6.3.5).
- 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. 3.3.1).
- gli identificatori per il controllo di sessione: il *process group-ID* e il *session id* ed il terminale di controllo (vedi sez. 10.1.2).
- la directory di lavoro e la directory radice (vedi sez. 5.1.7 e sez. 5.3.10).
- la maschera dei permessi di creazione (vedi sez. 5.3.7).
- la maschera dei segnali bloccati (vedi sez. 9.4.4) e le azioni installate (vedi sez. 9.3.1).
- i segmenti di memoria condivisa agganciati al processo (vedi sez. 12.2.6).
- i limiti sulle risorse (vedi sez. 8.3.2).
- le variabili di ambiente (vedi sez. 2.3.4).

le differenze fra padre e figlio dopo la `fork` invece sono:

- il valore di ritorno di `fork`.
- il *pid* (*process id*).
- il *ppid* (*parent process id*), quello del figlio viene impostato al *pid* del padre.
- i valori dei tempi di esecuzione della struttura `tms` (vedi sez. 8.4.2) che nel figlio sono posti a zero.
- i *lock* sui file (vedi sez. 11.4), che non vengono ereditati dal figlio.
- gli allarmi ed i segnali pendenti (vedi sez. 9.3.1), che per il figlio vengono cancellati.

### 3.2.3 La funzione `vfork`

La funzione `vfork` 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`), è deprecato; per questo eviteremo di trattarla ulteriormente.

### 3.2.4 La conclusione di un processo.

In sez. 2.1.2 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. 2.1.2 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), il ritorno dalla funzione `main` (equivalente alla chiamata di `exit`), e la chiamata ad `_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` per invocare una chiusura anomala, o essere terminato da un segnale. 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: 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 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. 9.3.6).
- 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. 10.1.3).
- 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. 10.1.3).

Oltre queste operazioni è però necessario poter disporre di un meccanismo ulteriore che consente 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. 2.1.2 che lo stato di uscita del processo viene caratterizzato tramite il valore del cosiddetto *exit status*, cioè il valore passato alle funzioni `exit` o `_exit` (o dal 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. 3.2.5) 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 ha 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`. 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 viene sostituito con il *pid* di `init` (e cioè con 1); in questo modo ogni processo avrà sempre un padre (nel caso possiamo parlare di un padre *adattivo*) 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 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. 8.4.1) 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. 3.5). Quando il padre effettuerà la lettura dello stato di uscita anche questa informazione, non più necessaria, verrà scartata e la terminazione potrà dirsi completamente conclusa.

Possiamo utilizzare il nostro programma di prova per analizzare anche questa condizione: lanciamo il comando `forktest` in background, indicando al processo padre di 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 stati terminati.

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 figli. In questo caso si deve sempre avere cura di far leggere l'eventuale stato di uscita di tutti i figli (in genere questo si fa attraverso un apposito *signal handler*, che chiama la funzione `wait`, vedi sez. 9.3.6 e sez. 3.2.5). Questa operazione è necessaria perché anche se gli *zombie* non consumano risorse di memoria o processore, occupano comunque una voce nella tabella dei processi, che a lungo andare potrebbe esaurirsi.

Si noti che quando un processo adottato da `init` termina, esso non diviene uno *zombie*; questo perché una delle funzioni di `init` è appunto quella di chiamare la funzione `wait` per i processi 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: alla sua terminazione infatti tutti i suoi figli (compresi gli *zombie*) verranno adottati da `init`, il quale provvederà a completarne la terminazione.

Si tenga presente infine che siccome gli *zombie* sono processi già usciti, non c'è modo di eliminarli con il comando `kill`; l'unica possibilità di cancellarli dalla tabella dei processi è quella di terminare il processo che li ha generati, in modo che `init` possa adottarli e provvedere a concluderne la terminazione.

### 3.2.5 Le funzioni `wait` e `waitpid`

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; le funzioni deputate a questo compito sono sostanzialmente due, `wait` e `waitpid`. La prima, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
    Sospende il processo corrente finché un figlio non è uscito, o finché un segnale termina il
    processo o chiama una funzione di gestione.
```

La funzione restituisce il *pid* del figlio in caso di successo e -1 in caso di errore; `errno` può assumere i valori:

`EINTR` la funzione è stata interrotta da un segnale.

è presente fin dalle prime versioni di Unix; la funzione ritorna non appena un processo figlio termina. Se un figlio è già terminato la funzione ritorna immediatamente, se più di un figlio è terminato occorre chiamare la funzione più volte se si vuole recuperare lo stato di terminazione di tutti quanti.

Al ritorno della funzione lo stato di terminazione del figlio viene salvato nella variabile punta da **status** e tutte le risorse del kernel relative al processo (vedi sez. 3.2.4) vengono rilasciate. Nel caso un processo abbia più figli il valore di ritorno (il *pid* del figlio) permette di identificare qual'è quello che è uscito.

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 un processo specifico occorrerebbe predisporre un meccanismo che tenga conto dei processi già terminati, e provvedere a ripetere la chiamata alla funzione nel caso il processo cercato sia ancora attivo.

Per questo motivo lo standard POSIX.1 ha introdotto la funzione **waitpid** che effettua lo stesso servizio, ma dispone di una serie di funzionalità più ampie, legate anche al controllo di sessione (si veda sez. 10.1). Dato che è possibile ottenere lo stesso comportamento di **wait** si consiglia di utilizzare sempre questa funzione, il cui prototipo è:

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

Attende la conclusione di un processo figlio.

La funzione restituisce il *pid* del processo che è uscito, 0 se è stata specificata l'opzione **WNOHANG** e il processo non è uscito e -1 per un errore, nel qual caso **errno** assumerà i valori:

<b>EINTR</b>	se non è stata specificata l'opzione <b>WNOHANG</b> e la funzione è stata interrotta da un segnale.
<b>ECHILD</b>	il processo specificato da <b>pid</b> non esiste o non è figlio del processo chiamante.

Le differenze principali fra le due funzioni sono che **wait** si blocca sempre fino a che un processo figlio non termina, mentre **waitpid** ha la possibilità di specificare un'opzione **WNOHANG** che ne previene il blocco; inoltre **waitpid** può specificare in maniera flessibile quale processo attendere, sulla base del valore fornito dall'argomento **pid**, secondo lo specchietto riportato in tab. 3.1.

Valore	Opzione	Significato
< -1	-	attende per un figlio il cui <i>process group</i> (vedi sez. 10.1.2) è uguale al valore assoluto di <b>pid</b> .
-1	<b>WAIT_ANY</b>	attende per un figlio qualsiasi, usata in questa maniera è equivalente a <b>wait</b> .
0	<b>WAIT_MYPGRP</b>	attende per un figlio il cui <i>process group</i> è uguale a quello del processo chiamante.
> 0	-	attende per un figlio il cui <b>pid</b> è uguale al valore di <b>pid</b> .

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

Il comportamento di **waitpid** può inoltre essere modificato passando delle opportune opzioni tramite l'argomento **option**. I valori possibili sono il già citato **WNOHANG**, che previene il blocco della funzione quando il processo figlio non è terminato, e **WUNTRACED** che permette di tracciare i processi bloccati. Il valore dell'opzione deve essere specificato come maschera binaria ottenuta con l'OR delle suddette costanti con zero.

In genere si utilizza **WUNTRACED** all'interno del controllo di sessione, (l'argomento è trattato in sez. 10.1). In tal caso infatti la funzione ritorna, restituendone il *pid*, quando c'è un processo figlio che è entrato in stato di sleep (vedi tab. 3.5) e del quale non si è ancora letto lo stato (con questa

stessa opzione). In Linux sono previste altre opzioni non standard relative al comportamento con i thread, che riprenderemo in sez. ??.

La terminazione di un processo figlio 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 delle azioni prese dal kernel alla conclusione di un processo è quella di mandare un segnale di **SIGCHLD** al padre. L'azione predefinita (si veda sez. 9.1.1) per questo segnale 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.

In genere in un programma non si vuole essere forzati ad attendere la conclusione di un processo per proseguire, specie se tutto questo serve solo per leggerne lo stato di chiusura (ed evitare la presenza di *zombie*), per questo la modalità più usata per 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. 9.4.1). In questo caso infatti, dato che il segnale è generato dalla terminazione di un figlio, avremo la certezza che la chiamata a **wait** non si bloccherà.

Macro	Descrizione
<b>WIFEXITED(s)</b>	Condizione vera (valore non nullo) per un processo figlio che sia terminato normalmente.
<b>WEXITSTATUS(s)</b>	Restituisce gli otto bit meno significativi dello stato di uscita del processo (passato attraverso <b>_exit</b> , <b>exit</b> o come valore di ritorno di <b>main</b> ). Può essere valutata solo se <b>WIFEXITED</b> ha restituito un valore non nullo.
<b>WIFSIGNALED(s)</b>	Vera se il processo figlio è terminato in maniera anomala a causa di un segnale che non è stato catturato (vedi sez. 9.1.4).
<b>WTERMSIG(s)</b>	restituisce il numero del segnale che ha causato la terminazione anomala del processo. Può essere valutata solo se <b>WIFSIGNALED</b> ha restituito un valore non nullo.
<b>WCOREDUMP(s)</b>	Vera se il processo terminato ha generato un file si <i>core dump</i> . Può essere valutata solo se <b>WIFSIGNALED</b> ha restituito un valore non nullo. <sup>8</sup>
<b>WIFSTOPPED(s)</b>	Vera se il processo che ha causato il ritorno di <b>waitpid</b> è bloccato. L'uso è possibile solo avendo specificato l'opzione <b>WUNTRACED</b> .
<b>WSTOPSIG(s)</b>	restituisce il numero del segnale che ha bloccato il processo, Può essere valutata solo se <b>WIFSTOPPED</b> ha restituito un valore non nullo.

**Tabella 3.2:** Descrizione delle varie macro di preprocessore utilizzabili per verificare lo stato di terminazione **s** di un processo.

Entrambe le funzioni di attesa restituiscono lo stato di terminazione del processo tramite il puntatore **status** (se non interessa memorizzare lo stato si può passare un puntatore nullo). Il valore restituito da entrambe le funzioni dipende dall'implementazione, e tradizionalmente alcuni bit (in genere 8) sono riservati per memorizzare lo stato di uscita, e altri per indicare il segnale che ha causato la terminazione (in caso di conclusione anomala), uno per indicare se è stato generato un core file, ecc.<sup>9</sup>

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. 3.2 (si tenga presente che queste macro prendono come parametro la variabile di tipo **int** puntata da **status**).

Si tenga conto che nel caso di conclusione anomala il valore restituito da **WTERMSIG** può essere confrontato con le costanti definite in **signal.h** ed elencate in tab. 9.3, e stampato usando le apposite funzioni trattate in sez. 9.2.9.

<sup>8</sup>questa macro non è definita dallo standard POSIX.1, ma è presente come estensione sia in Linux che in altri Unix.

<sup>9</sup>le 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>**.

### 3.2.6 Le funzioni wait3 e wait4

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 parametro attraverso il quale il kernel può restituire al padre informazioni sulle risorse usate dal processo terminato e dai vari figli. Le due funzioni sono `wait3` e `wait4`, che diventano accessibili definendo la macro `_USE_BSD`; i loro prototipi sono:

```
#include <sys/times.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/resource.h>
pid_t wait4(pid_t pid, int * status, int options, struct rusage * rusage)
    È identica a waitpid sia per comportamento che per i valori dei parametri, ma restituisce
    in rusage un sommario delle risorse usate dal processo.
pid_t wait3(int *status, int options, struct rusage *rusage)
    Prima versione, equivalente a wait4(-1, &status, opt, rusage) è ormai deprecata in
    favore di wait4.
```

la struttura `rusage` è definita in `sys/resource.h`, e viene utilizzata anche dalla funzione `getrusage` (vedi sez. 8.3.1) per ottenere le risorse di sistema usate da un processo; la sua definizione è riportata in fig. 8.6.

### 3.2.7 Le funzioni exec

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, lo heap, i dati ed il testo del processo corrente con un nuovo programma letto da disco.

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), sono tutte un front-end a `execve`. Il prototipo di quest'ultima è:

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

La funzione ritorna solo in caso di errore, restituendo -1; nel qual caso `errno` può assumere i valori:

<b>EACCES</b>	il file non è eseguibile, oppure il filesystem è montato in <code>noexec</code> , oppure non è un file regolare o un interprete.
<b>EPERM</b>	il file ha i bit <code>suid</code> o <code>sgid</code> , l'utente non è root, il processo viene tracciato, o il filesystem è montato con l'opzione <code>nosuid</code> .
<b>ENOEXEC</b>	il file è in un formato non eseguibile o non riconosciuto come tale, o compilato per un'altra architettura.
<b>ENOENT</b>	il file o una delle librerie dinamiche o l'interprete necessari per eseguirlo non esistono.
<b>ETXTBSY</b>	L'eseguibile è aperto in scrittura da uno o più processi.
<b>EINVAL</b>	L'eseguibile ELF ha più di un segmento <code>PF_INTERP</code> , cioè chiede di essere eseguito da più di un interprete.
<b>ELIBBAD</b>	Un interprete ELF non è in un formato riconoscibile.
<b>E2BIG</b>	La lista degli argomenti è troppo grande.

ed inoltre anche `EFAULT`, `ENOMEM`, `EIO`, `ENAMETOOLONG`, `ELOOP`, `ENOTDIR`, `ENFILE`, `EMFILE`.

La funzione `exec` esegue il file o lo script indicato da `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[])`.

Le altre funzioni della famiglia servono per fornire all'utente una serie possibile di diverse interfacce per la creazione di un nuovo processo. 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[])
Sostituiscono l'immagine corrente del processo con quella indicata nel primo argomento. I parametri successivi consentono di specificare gli argomenti a linea di comando e l'ambiente ricevuti dal nuovo processo.

Queste funzioni ritornano solo in caso di errore, restituendo -1; nel qual caso errno assumerà i valori visti in precedenza per execve.
```

Per capire meglio le differenze fra le funzioni della famiglia si può fare riferimento allo specchietto riportato in tab. 3.3. La prima differenza riguarda le modalità di passaggio dei parametri 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:

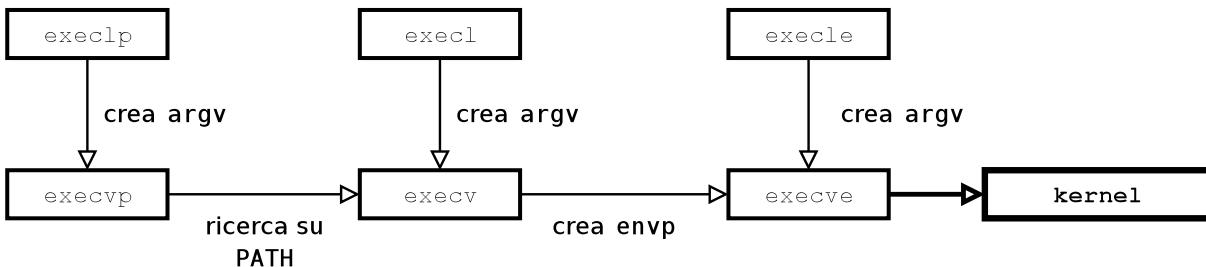
```
char *arg0, char *arg1, ..., char *argn, NULL
```

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.

Caratteristiche	Funzioni					
	execl	execp	execle	execv	execvp	execve
argomenti a lista	•	•	•			
argomenti a vettore				•	•	•
filename completo	•	•	•	•	•	•
ricerca su PATH				•		
ambiente a vettore	•	•	•	•	•	•
uso di <code>environ</code>						

**Tabella 3.3:** Confronto delle caratteristiche delle varie funzioni della famiglia `exec`.

La seconda differenza fra le funzioni riguarda le modalità con cui si specifica il programma che si vuole eseguire. Con lo mnemonic `p` si indicano le due funzioni che replicano il comportamento della shell nello specificare il comando da eseguire; quando il parametro `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`.



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

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. 2.3.4) del processo di partenza per costruire l'ambiente.

Oltre a mantenere lo stesso *pid*, il nuovo programma fatto partire da `exec` assume anche una serie di altre proprietà del processo chiamante; la lista completa 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. 3.3.1).
- il *session id* (*sid*) ed il *process group-ID* (*pgid*), vedi sez. 10.1.2.
- il terminale di controllo (vedi sez. 10.1.3).
- il tempo restante ad un allarme (vedi sez. 9.3.4).
- la directory radice e la directory di lavoro corrente (vedi sez. 5.1.7).
- la maschera di creazione dei file (`umask`, vedi sez. 5.3.7) ed i *lock* sui file (vedi sez. 11.4).
- i segnali sospesi (*pending*) e la maschera dei segnali (si veda sez. 9.4.4).
- i limiti sulle risorse (vedi sez. 8.3.2).
- i valori delle variabili `tms_utime`, `tms_stime`, `tms_cutime`, `tms ustime` (vedi sez. 8.4.2).

Inoltre i segnali che sono stati impostati per essere ignorati nel processo chiamante mantengono la stessa impostazione pure nel nuovo programma, tutti gli altri segnali vengono impostati alla loro azione predefinita. Un caso speciale è il segnale `SIGCHLD` che, quando impostato a `SIG_IGN`, può anche non essere reimpostato a `SIG_DFL` (si veda sez. 9.3.1).

La gestione dei file aperti dipende dal valore che ha il flag di *close-on-exec* (vedi anche sez. 6.3.5) per ciascun file descriptor. 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 una chiamata esplicita a `fcntl` che imposti 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. 5.1.6) che effettua da sola l'impostazione del flag di *close-on-exec* sulle directory che apre, in maniera trasparente all'utente.

Abbiamo detto che l'*user-ID reale* ed il *group-ID reale* restano gli stessi all'esecuzione di `exec`; lo stesso vale per l'*user-ID effettivo* ed il *group-ID effettivo* (il significato di questi identificatori è trattato in sez. 3.3.1), tranne quando il file che si va ad eseguire abbia o il *suid* bit o lo *sgid* bit impostato, in questo caso l'*user-ID effettivo* ed il *group-ID effettivo* vengono impostati rispettivamente all'utente o al gruppo cui il file appartiene (per i dettagli vedi sez. 3.3).

Se il file da eseguire è in formato *a.out* e necessita di librerie condivise, viene lanciato il *linker* dinamico `ld.so` prima del programma per caricare le librerie necessarie ed effettuare il link dell'eseguibile. Se il programma è in formato ELF per caricare le librerie dinamiche viene usato l'interprete indicato nel segmento `PT_INTERP`, in genere questo è `/lib/ld-linux.so.1` per

programmi linkati con le *libc5*, e */lib/ld-linux.so.2* per programmi linkati con le *glibc*. Infine nel caso il file sia uno script esso deve iniziare con una linea nella forma `#!/path/to/interpreter` dove l'interprete indicato deve esse un programma valido (binario, non un altro script) che verrà chiamato come se si fosse eseguito il comando `interpreter [argomenti] filename`.

Con la famiglia delle `exec` si chiude il novero delle funzioni su cui è basata la gestione 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.

### 3.3 Il controllo di accesso

In questa sezione esamineremo le problematiche relative al controllo di accesso dal punto di vista del 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.

#### 3.3.1 Gli identificatori del controllo di accesso

Come accennato in sez. 1.1.5 il modello base<sup>10</sup> 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 ed il group-ID; 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. 5.3 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 ad un utente e ad 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 gli Unix prevedono che i processi abbiano almeno due gruppi di identificatori, chiamati rispettivamente *real* ed *effective* (cioè *reali* ed *effettivi*). Nel caso di Linux si aggiungono poi altri due gruppi, il *saved* (salvati) ed il *filesystem* (di *filesystem*), secondo la situazione illustrata in tab. 3.4.

Al primo gruppo appartengono l'*user-ID reale* ed il *group-ID 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. 3.3.2) 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.

---

<sup>10</sup>in realtà già esistono estensioni di questo modello base, che lo rendono più flessibile e controllabile, come le *capabilities*, le ACL per i file 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, il *Linux Security Modules*, o LSM, in grado di fornire diversi agganci a livello del kernel per modularizzare tutti i possibili controlli di accesso.

Suffisso	Gruppo	Denominazione	Significato
<i>uid</i> <i>gid</i>	<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
<i>euid</i> <i>egid</i> –	<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>	è una copia dell' <i>euid</i> iniziale è una copia dell' <i>egid</i> iniziale
<i>fsuid</i> <i>fsgid</i>	<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 3.4:** Identificatori di utente e gruppo associati a ciascun processo con indicazione dei suffissi usati dalle varie funzioni di manipolazione.

Al secondo gruppo appartengono lo *user-ID effettivo* ed il *group-ID effettivo* (a cui si aggiungono gli eventuali *group-ID supplementari* dei gruppi dei quali l'utente fa parte). Questi sono invece gli identificatori usati nella verifiche dei permessi del processo e per il controllo di accesso ai file (argomento affrontato in dettaglio in sez. 5.3.1).

Questi identificatori normalmente sono identici ai corrispondenti del gruppo *real* tranne nel caso in cui, come accennato in sez. 3.2.7, 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. 5.3.2). In questo caso essi saranno impostati all'utente e al gruppo proprietari del file. Questo consente, per programmi in cui ci sia necessità, di dare a qualunque utente normale privilegi o permessi di un altro (o dell'amministratore).

Come nel caso del *pid* e del *ppid*, anche tutti questi identificatori possono essere letti attraverso le rispettive funzioni: *getuid*, *geteuid*, *getgid* e *getegid*, i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
uid_t getuid(void)
    Restituisce l'user-ID reale del processo corrente.
uid_t geteuid(void)
    Restituisce l'user-ID effettivo del processo corrente.
gid_t getgid(void)
    Restituisce il group-ID reale del processo corrente.
gid_t getegid(void)
    Restituisce il group-ID effettivo del processo corrente.
```

Queste funzioni non riportano condizioni di errore.

In generale l'uso di privilegi superiori 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 la costante *\_POSIX\_SAVED\_IDS*,<sup>11</sup> il secondo gruppo è specifico di Linux e viene usato per migliorare la sicurezza con NFS.

L'*user-ID salvato* ed il *group-ID salvato* sono copie dell'*user-ID effettivo* e del *group-ID effettivo* del processo padre, e vengono impostati dalla funzione *exec* all'avvio del processo, come copie dell'*user-ID effettivo* e del *group-ID effettivo* dopo che questi sono stati impostati tenendo conto di eventuali *suid* o *sgid*. Essi quindi consentono di tenere traccia di quale fossero utente e gruppo effettivi all'inizio dell'esecuzione di un nuovo programma.

<sup>11</sup>in 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.

L'*user-ID* di filesystem e il *group-ID* di filesystem sono un'estensione introdotta in Linux per rendere più sicuro l'uso di NFS (torneremo sull'argomento in sez. 3.3.6). 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. 5.3.1). 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.

### 3.3.2 Le funzioni setuid e setgid

Le due funzioni che vengono usate per cambiare identità (cioè utente e gruppo di appartenenza) ad un processo sono rispettivamente **setuid** e **setgid**; come accennato in sez. 3.3.1 in Linux esse seguono la semantica POSIX che prevede l'esistenza dell'*user-ID salvato* e del *group-ID salvato*; i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int setuid(uid_t uid)
    Imposta l'user-ID del processo corrente.
int setgid(gid_t gid)
    Imposta il group-ID del processo corrente.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore possibile è **EPERM**.

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 *group-ID* invece che all'*user-ID*. Gli eventuali *group-ID supplementari* non vengono modificati.

L'effetto della chiamata è diverso a seconda dei privilegi del processo; se l'*user-ID effettivo* è zero (cioè è quello dell'amministratore di sistema) allora tutti gli identificatori (*real*, *effective* e *saved*) vengono impostati al valore specificato da *uid*, altrimenti viene impostato solo l'*user-ID effettivo*, e soltanto se il valore specificato corrisponde o all'*user-ID reale* o all'*user-ID salvato*. Negli altri casi viene segnalato un errore (con **EPERM**).

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. 5.3.2) di riportare l'*user-ID 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/log/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 (*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:

<i>group-ID reale</i>	=	<i>gid</i> (del chiamante)
<i>group-ID effettivo</i>	=	<i>utmp</i>
<i>group-ID salvato</i>	=	<i>utmp</i>

In questo modo, dato che il *group-ID effettivo* è quello giusto, il programma può accedere a **/var/log/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 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/log/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/log/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 `login` una volta che crea una nuova shell per l'utente; ma quando si vuole cambiare soltanto l'*user-ID* effettivo del processo per cedere i privilegi occorre ricorrere ad altre funzioni (si veda ad esempio sez. 3.3.4).

### 3.3.3 Le funzioni `setreuid` e `setregid`

Le due funzioni `setreuid` e `setregid` derivano da BSD che, non supportando<sup>12</sup> 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 l'user-ID reale e l'user-ID effettivo del processo corrente ai valori specificati da
    ruid e euid.
int setregid(gid_t rgid, gid_t egid)
    Imposta il group-ID reale ed il group-ID effettivo del processo corrente ai valori specificati
    da rgid e egid.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore possibile è `EPERM`.

La due funzioni sono analoghe ed il loro comportamento è identico; quanto detto per la prima riguardo l'*user-ID*, si applica immediatamente alla seconda per il *group-ID*. I processi non privilegiati possono impostare solo i valori del loro *user-ID* effettivo o reale; 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 *user-ID* reale e effettivo, e pertanto è possibile implementare un comportamento simile a quello visto in precedenza per `setgid`,

<sup>12</sup>almeno fino alla versione 4.3+BSD TODO, FIXME verificare e aggiornare la nota.

cedendo i privilegi con un primo scambio, e recuperandoli, 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 user-ID 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'user-ID reale a quello effettivo) in caso contrario il nuovo programma potrebbe a sua volta effettuare uno scambio e riottenere privilegi non previsti.

Lo stesso problema di propagazione dei privilegi ad eventuali processi figli si pone per l'user-ID salvato: questa funzione deriva da un'implementazione 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'user-ID reale corrente, l'user-ID salvato viene automaticamente uniformato al valore dell'user-ID effettivo.

### 3.3.4 Le funzioni `seteuid` e `setegid`

Le due funzioni `seteuid` e `setegid` sono un'estensione allo standard POSIX.1 (ma sono comunque supportate dalla maggior parte degli Unix) e vengono usate per cambiare gli identificatori del gruppo *effective*; i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int seteuid(uid_t uid)
    Imposta l'user-ID effettivo del processo corrente a uid.
int setegid(gid_t gid)
    Imposta il group-ID effettivo del processo corrente a gid.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore è `EPERM`.

Come per le precedenti le due funzioni sono identiche, per cui tratteremo solo la prima. Gli utenti normali possono impostare l'user-ID effettivo solo al valore dell'user-ID reale o dell'user-ID salvato, l'amministratore può specificare qualunque valore. Queste funzioni sono usate per permettere all'amministratore di impostare solo l'user-ID effettivo, dato che l'uso normale di `setuid` comporta l'impostazione di tutti gli identificatori.

### 3.3.5 Le funzioni `setresuid` e `setresgid`

Le due funzioni `setresuid` e `setresgid` sono un'estensione introdotta in Linux,<sup>13</sup> 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)
    Imposta l'user-ID reale, l'user-ID effettivo e l'user-ID salvato del processo corrente ai valori specificati rispettivamente da ruid, euid e suid.
int setresgid(gid_t rgid, gid_t egid, gid_t sgid)
    Imposta il group-ID reale, il group-ID effettivo ed il group-ID salvato del processo corrente ai valori specificati rispettivamente da rgid, egid e sgid.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore è `EPERM`.

Le due funzioni sono identiche, quanto detto per la prima riguardo gli user-ID si applica alla seconda per i group-ID. I processi non privilegiati possono cambiare uno qualunque degli user-ID solo ad un valore corrispondente o all'user-ID reale, o a quello effettivo o a quello salvato,

<sup>13</sup>a partire dal kernel 2.1.44.

l'amministratore può specificare i valori che vuole; un valore di -1 per un qualunque parametro lascia inalterato l'identificatore corrispondente.

Per queste funzioni esistono anche due controparti che permettono di leggere in blocco i vari identificatori: `getresuid` e `getresgid`; i loro prototipi sono:

```
#include <unistd.h>
#include <sys/types.h>
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid)
    Legge l'user-ID reale, l'user-ID effettivo e l'user-ID salvato del processo corrente.
int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid)
    Legge il group-ID reale, il group-ID effettivo e il group-ID salvato del processo corrente.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore possibile è `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*.

### 3.3.6 Le funzioni `setfsuid` e `setfsgid`

Queste funzioni 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. 3.3.1 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'user-ID effettivo o l'user-ID 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'user-ID 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 usate 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'user-ID di filesystem del processo corrente a fsuid.
int setfsgid(gid_t fsgid)
    Imposta il group-ID di filesystem del processo corrente a fsgid.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di fallimento: l'unico errore possibile è `EPERM`.

queste 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 di quelli del gruppo *real*, *effective* o *saved*.

### 3.3.7 Le funzioni `setgroups` e `getgroups`

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

supplementari<sup>14</sup> in aggiunta al gruppo primario; questi vengono ereditati dal processo padre e possono essere cambiati con queste funzioni.

La funzione 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 restituisce il numero di gruppi letti in caso di successo e -1 in caso di fallimento, nel qual caso `errno` assumerà i valori:

- |               |   |
|---------------|---|
| <b>EFAULT</b> | <code>list</code> non ha un indirizzo valido.   |
| <b>EINVAL</b> | il valore di <code>size</code> 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` di dimensione `size`. Non è specificato se la funzione inserisca o meno nella lista il group-ID effettivo del processo. Se si specifica un valore di `size` uguale a 0 `list` non viene modificato, ma si ottiene il numero di gruppi supplementari.

Una seconda funzione, `getgrouplist`, può invece essere usata per ottenere tutti i gruppi a cui appartiene un certo utente; il suo prototipo è:

```
#include <sys/types.h>
#include <grp.h>
int getgrouplist(const char *user, gid_t group, gid_t *groups, int *ngroups)
    Legge i gruppi supplementari.
```

La funzione legge fino ad un massimo di `ngroups` valori, restituisce 0 in caso di successo e -1 in caso di fallimento.

La funzione legge i gruppi supplementari dell'utente specificato da `user`, eseguendo una scansione del database dei gruppi (si veda sez. 8.2.3). Ritorna poi in `groups` la lista di quelli a cui l'utente appartiene. Si noti che `ngroups` è passato come puntatore perché, qualora il valore specificato sia troppo piccolo, la funzione ritorna -1, passando indietro il numero dei gruppi trovati.

Per impostare i gruppi supplementari di un processo ci sono due funzioni, che possono essere usate solo se si hanno i privilegi di amministratore. La prima delle due è `setgroups`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <grp.h>
int setgroups(size_t size, gid_t *list)
    Imposta i gruppi supplementari del processo.
```

La funzione restituisce 0 in caso di successo e -1 in caso di fallimento, nel qual caso `errno` assumerà i valori:

- |               |  |
|---------------|--|
| <b>EFAULT</b> | <code>list</code> non ha un indirizzo valido.                            |
| <b>EPERM</b>  | il processo non ha i privilegi di amministratore.                        |
| <b>EINVAL</b> | il valore di <code>size</code> è maggiore del valore massimo consentito. |

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 è un parametro di sistema, che può essere ricavato con le modalità spiegate in sez. 8.1.

<sup>14</sup>il numero massimo di gruppi secondari può essere ottenuto con `sysconf` (vedi sez. 8.1.2), leggendo il parametro `_SC_NGROUPS_MAX`.

Se invece si vogliono impostare i gruppi supplementari del processo a quelli di un utente specifico, si può usare `initgroups` il cui prototipo è:

```
#include <sys/types.h>
#include <grp.h>
int initgroups(const char *user, gid_t group)
    Inizializza la lista dei gruppi supplementari.
```

La funzione restituisce 0 in caso di successo e -1 in caso di fallimento, nel qual caso `errno` assumerà gli stessi valori di `setgroups` più `ENOMEM` quando non c'è memoria sufficiente per allocare lo spazio per informazioni dei gruppi.

La funzione esegue la scansione del database dei gruppi (usualmente `/etc/groups`) cercando i gruppi di cui è membro l'utente `user` 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.

## 3.4 La gestione della priorità di esecuzione

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.

### 3.4.1 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.<sup>15</sup> 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. 3.5; ma soltanto i processi che sono nello stato *Runnable* concorrono per l'esecuzione. Questo vuol dire che, qualunque sia la sua priorità,

<sup>15</sup>nei 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.

un processo non potrà mai essere messo in esecuzione fintanto che esso si trova in uno qualunque degli altri stati.

Stato	STAT	Descrizione
<b>Runnable</b>	R	Il processo è in esecuzione o è pronto ad essere eseguito (cioè è in attesa che gli venga assegnata la CPU).
<b>Sleep</b>	S	Il processo è in attesa di un risposta dal sistema, ma può essere interrotto da un segnale.
<b>Uninterruptible Sleep</b>	D	Il processo è in attesa di un risposta dal sistema (in genere per I/O), e non può essere interrotto in nessuna circostanza.
<b>Stopped</b>	T	Il processo è stato fermato con un <b>SIGSTOP</b> , o è tracciato.
<b>Zombie</b>	Z	Il processo è terminato ma il suo stato di terminazione non è ancora stato letto dal padre.

**Tabella 3.5:** 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 (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. 3.4.2) è sempre stato basato su delle *priorità dinamiche*, in modo da assicurare che tutti i processi, anche i meno importanti, possano 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 sistemi real-time,<sup>16</sup> 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. 3.4.3.

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.

### 3.4.2 Il meccanismo di *scheduling* standard

A meno che non si abbiano esigenze specifiche, 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 tutti i processi ordinari hanno la stessa priorità assoluta. Quello che determina quale, fra tutti i processi in attesa di esecuzione, sarà eseguito per primo, è la priorità dinamica, che è chiamata così proprio perché varia nel corso dell'esecuzione di un

<sup>16</sup>per 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 sfornamento dei tempi avrebbe conseguenze disastrose), e *soft-real-time* in cui un occasionale sfornamento è ritenuto accettabile.

processo. Oltre a questo la priorità dinamica determina quanto a lungo un processo continuerà ad essere eseguito, e quando un processo potrà subentrare ad un altro nell'esecuzione.

Il meccanismo usato da Linux è piuttosto semplice, ad ogni processo è assegnata una *time-slice*, cioè un intervallo di tempo (letteralmente una fetta) per il quale esso deve essere eseguito. Il valore della *time-slice* è controllato dalla cosiddetta *nice* (o *niceness*) del processo. Essa è contenuta nel campo **nice** di **task\_struct**; tutti i processi vengono creati con lo stesso valore, ed essa specifica il valore della durata iniziale della *time-slice* che viene assegnato ad un altro campo della struttura (**counter**) quando il processo viene eseguito per la prima volta e diminuito progressivamente ad ogni interruzione del timer.

Durante la sua esecuzione lo scheduler scandisce la coda dei processi in stato *runnable* associando, in base al valore di **counter**, un peso ad ogni processo in attesa di esecuzione,<sup>17</sup> chi ha il peso più alto verrà posto in esecuzione, ed il precedente processo sarà spostato in fondo alla coda. Dato che ad ogni interruzione del timer il valore di **counter** del processo corrente viene diminuito, questo assicura che anche i processi con priorità più bassa verranno messi in esecuzione.

La priorità di un processo è così controllata attraverso il valore di **nice**, che stabilisce la durata della *time-slice*; per il meccanismo appena descritto infatti un valore più lungo assicura una maggiore attribuzione di CPU. 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 lo stesso valore di **nice** (nullo) e nessuno è privilegiato rispetto agli altri; il valore può essere modificato solo attraverso la funzione **nice**, il cui prototipo è:

```
#include <unistd.h>
int nice(int inc)
    Aumenta il valore di nice per il processo corrente.
```

La funzione ritorna zero in caso di successo e -1 in caso di errore, nel qual caso **errno** può assumere i valori:

**EPERM** un processo senza i privilegi di amministratore ha specificato un valore di **inc** negativo.

L'argomento **inc** indica l'incremento del valore di **nice**: quest'ultimo può assumere valori compresi fra **PRIOR\_MIN** e **PRIOR\_MAX** (che nel caso di Linux sono -19 e 20), 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, ogni utente può solo innalzare il valore di un suo processo. Solo l'amministratore può specificare valori negativi che permettono di aumentare la priorità di un processo.

In SUSv2 la funzione ritorna il nuovo valore di **nice**; Linux non segue questa convenzione, e per leggere il nuovo valore occorre invece usare la funzione **getpriority**, derivata da BSD, il cui prototipo è:

```
#include <sys/resource.h>
int getpriority(int which, int who)
    Restituisce il valore di nice per l'insieme dei processi specificati.
```

La funzione ritorna la priorità in caso di successo e -1 in caso di errore, nel qual caso **errno** può assumere i valori:

**ESRCH** non c'è nessun processo che corrisponda ai valori di **which** e **who**.
**EINVAL** il valore di **which** non è valido.

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.

<sup>17</sup>il calcolo del peso in realtà è un po' più complicato, ad esempio nei sistemi multiprocessore viene favorito un processo eseguito sulla stessa CPU, e a parità del valore di **counter** viene favorito chi ha una priorità più elevata.

La funzione permette, a seconda del valore di `which`, di leggere la priorità di un processo, di un gruppo di processi (vedi sez. 10.1.2) o di un utente, specificando un corrispondente valore per `who` secondo la legenda di tab. 3.6; un valore nullo di quest'ultimo indica il processo, il gruppo di processi o l'utente correnti.

<code>which</code>	<code>who</code>	Significato
<code>PRIOR_PROCESS</code>	<code>pid_t</code>	processo
<code>PRIOR_PRGR</code>	<code>pid_t</code>	process group
<code>PRIOR_USER</code>	<code>uid_t</code>	utente

**Tabella 3.6:** Legenda del valore dell'argomento `which` e del tipo dell'argomento `who` delle funzioni `getpriority` e `setpriority` per le tre possibili scelte.

La funzione restituisce la priorità più alta (cioè il valore più basso) fra quelle dei processi specificati; dato che -1 è un valore possibile, per poter rilevare una condizione di errore è necessario cancellare sempre `errno` prima della chiamata alla funzione, per verificare che essa resti uguale a zero.

Analogamente a `getpriority` la funzione `setpriority` permette di impostare la priorità di uno o più processi; il suo prototipo è:

```
#include <sys/resource.h>
int setpriority(int which, int who, int prio)
    Imposta la priorità per l'insieme dei processi specificati.
```

La funzione ritorna la priorità in caso di successo e -1 in caso di errore, nel qual caso `errno` può assumere i valori:

<code>ESRCH</code>	non c'è nessun processo che corrisponda ai valori di <code>which</code> e <code>who</code> .
<code>EINVAL</code>	il valore di <code>which</code> non è valido.
<code>EPERM</code>	un processo senza i privilegi di amministratore ha specificato un valore di <code>inc</code> negativo.
<code>EACCES</code>	un processo senza i privilegi di amministratore ha cercato di modificare la priorità di un processo di un altro utente.

La funzione impone la priorità al valore specificato da `prio` per tutti i processi indicati dagli argomenti `which` e `who`. La gestione dei permessi dipende dalle varie implementazioni; in Linux, secondo le specifiche dello standard SUSv3, e come avviene per tutti i sistemi che derivano da SysV, è richiesto che l'user-ID reale o effettivo del processo chiamante corrispondano al real user-ID (e solo quello) del processo di cui si vuole cambiare la priorità; per i sistemi derivati da BSD invece (SunOS, Ultrix, \*BSD) la corrispondenza può essere anche con l'user-ID effettivo.

### 3.4.3 Il meccanismo di *scheduling real-time*

Come spiegato in sez. 3.4.1 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,<sup>18</sup> 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. 2.2.7), il primo non è superabile e può comportare ritardi non prevedibili riguardo ai tempi di esecuzione di qualunque processo.

Occorre usare le priorità assolute 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

<sup>18</sup>questo 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*.

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 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 scelto; lo standard ne prevede due:

**FIFO** *First In First Out.* Il processo viene eseguito fintanto che non cede volontariamente la CPU, si blocca, finisce o viene interrotto da un processo a priorità più alta.

**RR** *Round Robin.* Ciascun processo viene eseguito a turno per un certo periodo di tempo (una *time slice*). Solo i processi con la stessa priorità ed in stato *runnable* entrano nel circolo.

La funzione per impostare le politiche di scheduling (sia real-time che ordinarie) ed i relativi parametri è `sched_setscheduler`; 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 la priorità in caso di successo e -1 in caso di errore, nel qual caso `errno` può assumere i valori:

<code>ESRCH</code>	il processo <code>pid</code> non esiste.
<code>EINVAL</code>	il valore di <code>policy</code> non esiste o il relativo valore di <code>p</code> non è valido.
<code>EPERM</code>	il processo non ha i privilegi per attivare la politica richiesta.

La funzione esegue l'impostazione per il processo specificato dall'argomento `pid`; un valore nullo esegue l'impostazione per il processo corrente. La politica di scheduling è specificata dall'argomento `policy` i cui possibili valori sono riportati in tab. 3.7; un valore negativo per `policy` mantiene la politica di scheduling corrente. Solo un processo con i privilegi di amministratore può impostare priorità assolute diverse da zero o politiche `SCHED_FIFO` e `SCHED_RR`.

Policy	Significato
<code>SCHED_FIFO</code>	Scheduling real-time con politica <i>FIFO</i>
<code>SCHED_RR</code>	Scheduling real-time con politica <i>Round Robin</i>
<code>SCHED_OTHER</code>	Scheduling ordinario

*Tabella 3.7:* Valori dell'argomento `policy` per la funzione `sched_setscheduler`.

Il valore della priorità è passato attraverso la struttura `sched_param` (riportata in fig. 3.5), il cui solo campo attualmente definito è `sched_priority`, che nel caso delle priorità assolute deve essere specificato nell'intervallo fra un valore massimo ed uno minimo, che nel caso sono rispettivamente 1 e 99 (il valore zero è legale, ma indica i processi normali).

Lo standard POSIX.1b prevede comunque che i due valori della massima e minima priorità statica possano essere ottenuti, per ciascuna delle politiche di scheduling realtime, tramite le due funzioni `sched_get_priority_max` e `sched_get_priority_min`, i cui prototipi sono:

---

```
struct sched_param {
    int sched_priority;
};
```

---

*Figura 3.5:* La struttura `sched_param`.

```
#include <sched.h>
int sched_get_priority_max(int policy)
    Legge il valore massimo della priorità statica per la politica di scheduling policy.
int sched_get_priority_min(int policy)
    Legge il valore minimo della priorità statica per la politica di scheduling policy.

La funzioni ritornano il valore della priorità in caso di successo e -1 in caso di errore, nel qual caso
errno può assumere i valori:
EINVAL    il valore di policy non è valido.
```

I processi con politica di scheduling **SCHED\_OTHER** devono specificare un valore nullo (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.

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 reinserito in coda alla lista); l'esecuzione viene ripresa subito solo nel caso che esso sia stato interrotto da un processo a priorità più alta.

La priorità assoluta può essere riletta indietro dalla funzione **sched\_getscheduler**, il cui prototipo è:

```
#include <sched.h>
int sched_getscheduler(pid_t pid)
    Legge la politica di scheduling per il processo pid.

La funzione ritorna la politica di scheduling in caso di successo e -1 in caso di errore, nel qual caso
errno può assumere i valori:
ESRCH    il processo pid non esiste.
EINVAL    il valore di pid è negativo.
```

La funzione restituisce il valore (secondo quanto elencato in tab. 3.7) della politica di scheduling per il processo specificato; se **pid** è nullo viene restituito quello del processo chiamante.

Se si intende operare solo sulla priorità assoluta di un processo si possono usare le funzioni **sched\_setparam** e **sched\_getparam**, i cui prototipi sono:

```
#include <sched.h>
int sched_setparam(pid_t pid, const struct sched_param *p)
    Imposta la priorità assoluta del processo pid.
int sched_getparam(pid_t pid, struct sched_param *p)
    Legge la priorità assoluta del processo pid.

La funzione ritorna la priorità in caso di successo e -1 in caso di errore, nel qual caso errno può
assumere i valori:
ESRCH    il processo pid non esiste.
EINVAL    il valore di pid è negativo.
```

L'uso di `sched_setparam` che è del tutto equivalente a `sched_setscheduler` con priority uguale a -1. Come per `sched_setscheduler` specificando 0 come valore di pid si opera sul processo corrente. La disponibilità di entrambe le funzioni può essere verificata controllando la macro `_POSIX_PRIORITY_SCHEDULING` che è definita nell'header `sched.h`.

L'ultima funzione 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:

```
#include <sched.h>
int sched_rr_get_interval(pid_t pid, struct timespec *tp)
    Legge in tp la durata della time slice per il processo pid.
```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` può assumere i valori:

<code>ESRCH</code>	il processo <code>pid</code> non esiste.
<code>ENOSYS</code>	la system call non è stata implementata.

La funzione restituisce il valore dell'intervallo di tempo usato per la politica *round robin* in una struttura `timespec`, (la cui definizione si può trovare in fig. 8.9).

Come accennato ogni processo che usa lo scheduling real-time può rilasciare volontariamente la CPU; questo viene fatto attraverso la funzione `sched_yield`, il cui prototipo è:

```
#include <sched.h>
int sched_yield(void)
    Rilascia volontariamente l'esecuzione.
```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` viene impostata opportunamente.

La funzione fa sì che il processo rilasci la CPU, in modo da essere rimesso in coda alla lista dei processi da eseguire, e permettere l'esecuzione di un altro processo; se però il processo è l'unico ad essere presente sulla coda l'esecuzione non sarà interrotta. In genere usano questa funzione i processi in modalità *fifo*, per permettere l'esecuzione degli altri processi con pari priorità quando la sezione più urgente è finita.

## 3.5 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.

Pur 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.

### 3.5.1 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. 3.5.2) 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. 12) o nelle operazioni con i file (vedremo alcuni esempi in sez. 6.3.2). 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. 9.4).

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. 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.

### 3.5.2 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 (le cosiddette *sezioni critiche*) del programma, siano opportunamente protette da meccanismi di sincronizzazione (torneremo su queste problematiche di questo tipo in cap. 12).

Un caso particolare di *race condition* sono poi i cosiddetti *deadlock*, 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. 3.5.1; questi problemi infatti possono essere risolti soltanto assicurandosi, quando essa sia richiesta, che sia possibile eseguire in maniera atomica le operazioni necessarie.

### 3.5.3 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 multi-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, e 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, le *glibc* però mettono 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

## L’architettura dei file

Uno dei concetti fondamentali dell’architettura di un sistema Unix è il cosiddetto *everything is a 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.

Questo significa che si può accedere a qualunque periferica del computer, dalla seriale, alla parallela, alla console, e agli stessi dischi attraverso i cosiddetti file di dispositivo (i *device file*). Questi sono dei file speciali agendo sui quali i programmi possono leggere, scrivere e compiere operazioni direttamente sulle periferiche, usando le stesse funzioni che si usano per i normali file di dati.

In questo capitolo forniremo una descrizione dell’architettura dei file in Linux, iniziando da una panoramica sulle caratteristiche principali delle interfacce con cui i processi accedono ai file (che tratteremo in dettaglio nei capitoli seguenti), per poi passare ad una descrizione più dettagliata delle modalità con cui detto accesso viene realizzato dal sistema.

### 4.1 L’architettura generale

Per poter accedere ai file, il kernel deve mettere a disposizione dei programmi le opportune interfacce che consentano di leggerne il contenuto; il sistema cioè deve provvedere ad organizzare e rendere accessibile in maniera opportuna l’informazione tenuta sullo spazio grezzo disponibile sui dischi. Questo viene fatto strutturando l’informazione sul disco attraverso quello che si chiama un *filesystem* (vedi sez. 4.2), essa poi viene resa disponibile ai processi attraverso quello che viene chiamato il *montaggio* del *filesystem*.

In questa sezione faremo una panoramica generica su come il sistema presenta i file ai processi, trattando l’organizzazione di file e directory, i tipi di file ed introducendo le interfacce disponibili e le loro caratteristiche.

#### 4.1.1 L’organizzazione di file e directory

In Unix, a differenza di quanto avviene in altri sistemi operativi, tutti i file vengono tenuti all’interno di un unico albero la cui radice (quella che viene chiamata *root directory*) viene montata all’avvio. Un file viene identificato dall’utente usando quello che viene chiamato *pathname*<sup>1</sup>, cioè il percorso che si deve fare per accedere al file a partire dalla *root directory*, che è composto da una serie di nomi separati da una /.

<sup>1</sup>il manuale della *glibc* depreca questa nomenclatura, che genererebbe confusione poiché *path* indica anche un insieme di directory su cui effettuare una ricerca (come quello in cui si cercano 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.

All'avvio del sistema, completata la fase di inizializzazione, il kernel riceve dal bootloader l'indicazione di quale dispositivo contiene il filesystem da usare come punto di partenza e questo viene montato come radice dell'albero (cioè nella directory `/`); tutti gli ulteriori filesystem che possono essere su altri dispositivi dovranno poi essere inseriti nell'albero montandoli su opportune directory del filesystem montato come radice.

Alcuni filesystem speciali (come `/proc` che contiene un'interfaccia ad alcune strutture interne del kernel) sono generati automaticamente dal kernel stesso, ma anche essi devono essere montati all'interno dell'albero dei file.

Una directory, come vedremo in maggior dettaglio in sez. 4.2.2, è anch'essa un file, solo che è un file particolare che il kernel riconosce come tale. Il suo scopo è quello di contenere una lista di nomi di file e le informazioni che associano ciascun nome al contenuto. Dato che questi nomi possono corrispondere ad un qualunque oggetto del filesystem, compresa un'altra directory, si ottiene naturalmente un'organizzazione ad albero inserendo directory in altre directory.

Un file può essere indicato rispetto alla directory corrente semplicemente specificandone il nome<sup>2</sup> da essa contenuto. All'interno dello stesso albero si potranno poi inserire anche tutti gli altri oggetti visti attraverso l'interfaccia che manipola i file come le *fifo*, i *link*, i *socket* e gli stessi file di dispositivo (questi ultimi, per convenzione, sono inseriti nella directory `/dev`).

Il nome completo di un file viene chiamato *pathname* ed il procedimento con cui si individua il file a cui esso fa riferimento è chiamato risoluzione del nome (*file name resolution* o *pathname resolution*). La risoluzione viene fatta esaminando il *pathname* da sinistra a destra e localizzando ogni nome nella directory indicata dal nome precedente usando `/` come separatore<sup>3</sup>: ovviamente, perché il procedimento funzioni, occorre che i nomi indicati come directory esistano e siano effettivamente directory, inoltre i permessi (si veda sez. 5.3) devono consentire l'accesso all'intero *pathname*.

Se il *pathname* comincia per `/` la ricerca parte dalla directory radice del processo; questa, a meno di un *chroot* (su cui torneremo in sez. 5.3.10) è la stessa per tutti i processi ed equivale alla directory radice dell'albero dei file: in questo caso si parla di un *pathname assoluto*. Altrimenti la ricerca parte dalla directory corrente (su cui torneremo in sez. 5.1.7) ed il *pathname* è detto *pathname relativo*.

I nomi `.` e `..` hanno un significato speciale e vengono inseriti in ogni directory: 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; nel caso la directory corrente coincida con la directory radice, allora il riferimento è a se stessa.

#### 4.1.2 I tipi di file

Come detto in precedenza, in Unix esistono vari tipi di file; in Linux questi sono implementati come oggetti del *Virtual File System* (vedi sez. 4.2.2) e sono presenti in tutti i filesystem unix-like utilizzabili con Linux. L'elenco dei vari tipi di file definiti dal *Virtual File System* è riportato in tab. 4.1.

Si tenga ben presente che questa classificazione non ha nulla a che fare con la classificazione dei file (che in questo caso sono sempre file di dati) in base al loro contenuto, o tipo di accesso. Essa riguarda invece il tipo di oggetti; in particolare è da notare la presenza dei cosiddetti file speciali. Alcuni di essi, come le *fifo* (che tratteremo in sez. 12.1.4) ed i *socket* (che tratteremo in cap. 14) non sono altro che dei riferimenti per utilizzare delle funzionalità di comunicazione fornite dal kernel. Gli altri sono i *file di dispositivo* (o *device file*) che costituiscono una interfaccia diretta per leggere e scrivere sui dispositivi fisici; essi vengono suddivisi in due grandi categorie,

<sup>2</sup>Il manuale delle *glibc* chiama i nomi contenuti nelle directory *componenti* (in inglese *file name components*), noi li chiameremo più semplicemente *nomi*.

<sup>3</sup>nel caso di nome vuoto, il costrutto `//` viene considerato equivalente a `/`.

a blocchi e a caratteri a seconda delle modalità in cui il dispositivo sottostante effettua le operazioni di I/O.<sup>4</sup>

Tipo di file		Descrizione
<i>regular file</i>	file regolare	un file che contiene dei dati (l'accezione normale di file)
<i>directory</i>	cartella o direttorio	un file che contiene una lista di nomi associati a degli <i>inode</i> (vedi sez. 4.2.1).
<i>symbolic link</i>	collegamento simbolico	un file che contiene un riferimento ad un altro file/directory
<i>char device</i>	dispositivo a caratteri	un file che identifica una periferica ad accesso a caratteri
<i>block device</i>	dispositivo a blocchi	un file che identifica una periferica ad accesso a blocchi
<i>fifo</i>	“coda”	un file speciale che identifica una linea di comunicazione software unidirezionale (vedi sez. 12.1.4).
<i>socket</i>	“presa”	un file speciale che identifica una linea di comunicazione software bidirezionale (vedi cap. 14)

**Tabella 4.1:** Tipologia dei file definiti nel VFS

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) né c’è una strutturazione a record per il cosiddetto “accesso diretto” come nel caso del VMS.<sup>5</sup>

Una seconda differenza è nel formato dei file ASCII: in Unix la fine riga è codificata in maniera diversa da Windows o Mac, in particolare il fine riga è il carattere LF (o \n) al posto del CR (\r) del Mac e del CR LF di Windows.<sup>6</sup> Questo può causare alcuni problemi qualora nei programmi si facciano assunzioni sul terminatore della riga.

Si ricordi infine che un kernel Unix non fornisce nessun supporto per la tipizzazione dei file di dati e che non c’è nessun supporto del sistema per le estensioni come parte del filesystem.<sup>7</sup> Ciò nonostante molti programmi adottano delle convenzioni per i nomi dei file, ad esempio il codice C normalmente si mette in file con l'estensione .c; un'altra tecnica molto usata è quella di utilizzare i primi 4 byte del file per memorizzare un *magic number* che classifichi il contenuto; entrambe queste tecniche, per quanto usate ed accettate in maniera diffusa, restano solo delle convenzioni il cui rispetto è demandato alle applicazioni stesse.

### 4.1.3 Le due interfacce ai file

In Linux le modalità di accesso ai file e le relative interfacce di programmazione sono due, basate su due diversi meccanismi con cui è possibile accedere al loro contenuto.

<sup>4</sup>in sostanza i dispositivi a blocchi (ad esempio i dischi) corrispondono a periferiche per le quali è richiesto che l'I/O venga effettuato per blocchi di dati di dimensioni fissate (ad esempio le dimensioni di un settore), mentre nei dispositivi a caratteri l'I/O viene effettuato senza nessuna particolare struttura.

<sup>5</sup>questo 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, operazioni di I/O direttamente sui dischi senza passare attraverso un filesystem (il cosiddetto *raw access*, introdotto coi kernel della serie 2.4.x).

<sup>6</sup>per questo esistono in Linux dei programmi come unix2dos e dos2unix che effettuano una conversione fra questi due formati di testo.

<sup>7</sup>non 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, come l'XFS della SGI, esiste la possibilità di associare delle risorse ai file, ma è una caratteristica tutt'ora poco utilizzata, dato che non corrisponde al modello classico dei file in un sistema Unix.

La prima è l’interfaccia standard di Unix, quella che il manuale delle *glibc* chiama interfaccia dei descrittori di file (o *file descriptor*). È un’interfaccia specifica dei sistemi unix-like e fornisce un accesso non bufferizzato; la tratteremo in dettaglio in cap. 6.

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

La seconda interfaccia è quella che il manuale della *glibc* chiama degli *stream*. Essa fornisce funzioni più evolute e un accesso bufferizzato (controllato dalla implementazione fatta dalle *glibc*), la tratteremo in dettaglio nel cap. 7.

Questa è l’interfaccia standard specificata dall’ANSI C e perciò si trova anche su tutti i sistemi non Unix. Gli *stream* sono oggetti complessi e sono rappresentati da puntatori ad un’opportuna struttura definita dalle librerie del C; si accede ad essi sempre in maniera indiretta utilizzando il tipo `FILE *`. L’interfaccia è definita nell’header `stdio.h`.

Entrambe le interfacce possono essere usate per l’accesso ai file come agli altri oggetti del VFS (fifo, socket, device, sui quali torneremo in dettaglio a tempo opportuno), ma per poter accedere alle operazioni di controllo (descritte in sez. 6.3.5 e sez. 6.3.6) 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. 11).

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 enormemente 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 operazioni di basso livello, o associare in un secondo tempo uno *stream* ad un *file descriptor*.

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.

## 4.2 L’architettura della gestione dei file

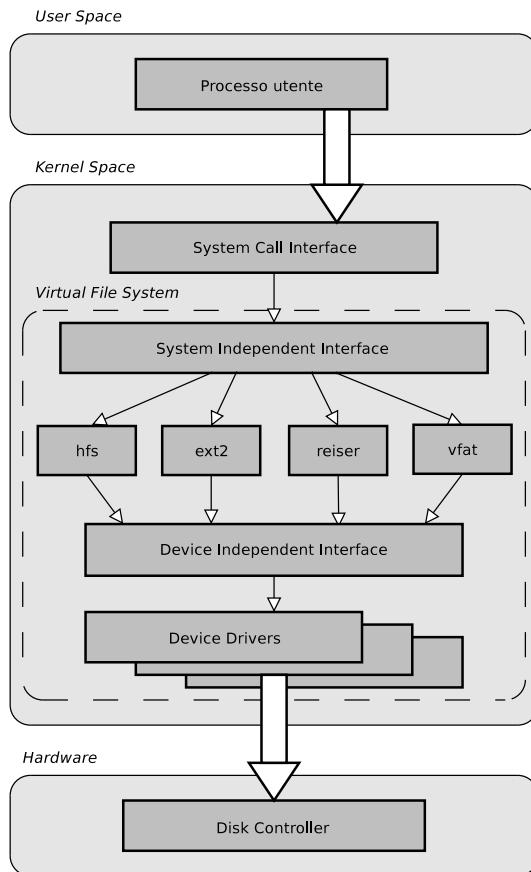
In questa sezione esamineremo come viene implementato l’accesso ai file in Linux, come il kernel può gestire diversi tipi di filesystem, descrivendo prima le caratteristiche generali di un filesystem di un sistema unix-like, per poi trattare in maniera un po’ più dettagliata il filesystem più usato con Linux, l’ext2.

### 4.2.1 Il *Virtual File System* di Linux

In Linux il concetto di *everything is a file* è stato implementato attraverso il *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. Esso fornisce un livello di indirezione che permette di collegare le operazioni di manipolazione sui file alle operazioni di I/O, e gestisce l’organizzazione di queste ultime nei vari modi in cui i diversi filesystem le

effettuano, permettendo la coesistenza di filesystem differenti all'interno dello stesso albero delle directory.

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 la chiamata alle opportune routine del filesystem specifico a cui si fa riferimento. Saranno queste a chiamare le funzioni di più basso livello che eseguono le operazioni di I/O sul dispositivo fisico, secondo lo schema riportato in fig. 4.1.



**Figura 4.1:** Schema delle operazioni del VFS.

Il VFS definisce un insieme di funzioni che tutti i filesystem devono implementare. L'interfaccia comprende tutte le funzioni che riguardano i file; le operazioni sono suddivise su tre tipi di oggetti: *filesystem*, *inode* e *file*, corrispondenti a tre apposite strutture definite nel kernel.

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` passandole un'apposita struttura `file_system_type` che contiene i dettagli per il riferimento all'implementazione del medesimo, che sarà aggiunta alla citata tabella.

In questo modo quando viene effettuata la richiesta di montare un nuovo disco (o qualunque altro *block device* che può contenere un filesystem), il VFS può ricavare dalla citata tabella il puntatore alle funzioni da chiamare nelle operazioni di montaggio. Quest'ultima è responsabile di leggere da disco il superblock (vedi sez. 4.2.4), inizializzare tutte le variabili interne e restituire uno speciale descrittore dei filesystem montati al VFS; attraverso quest'ultimo diventa possibile accedere alle routine specifiche per l'uso di quel filesystem.

Il primo oggetto usato dal VFS è il descrittore di filesystem, un puntatore ad una apposita struttura che contiene vari dati come le informazioni comuni ad ogni filesystem, i dati privati

relativi a quel filesystem specifico, e i puntatori alle funzioni del kernel relative al filesystem. Il VFS può così usare le funzioni contenute nel *filesystem descriptor* per accedere alle routine specifiche di quel filesystem.

Gli altri due descrittori usati dal VFS sono relativi agli altri due oggetti su cui è strutturata l'interfaccia. Ciascuno di essi contiene le informazioni relative al file in uso, insieme ai puntatori alle funzioni dello specifico filesystem usate per l'accesso dal VFS; in particolare il descrittore dell'*inode* contiene i puntatori alle funzioni che possono essere usate su qualunque file (come `link`, `stat` e `open`), mentre il descrittore di file contiene i puntatori alle funzioni che vengono usate sui file già aperti.

#### 4.2.2 Il funzionamento del *Virtual File System*

La funzione più importante implementata dal VFS è la system call `open` che permette di aprire un file. Dato un *pathname* viene eseguita una ricerca dentro la *directory entry cache* (in breve *dcache*), una tabella che contiene tutte le *directory entry* (in breve *dentry*) che permette di associare in maniera rapida ed efficiente il *pathname* a una specifica *dentry*.

Una singola *dentry* contiene in genere il puntatore ad un *inode*; quest'ultimo è la struttura base che sta sul disco e che identifica un singolo oggetto del VFS sia esso un file ordinario, una directory, un link simbolico, una FIFO, un file di dispositivo, o una qualsiasi altra cosa che possa essere rappresentata dal VFS (i tipi di file riportati in tab. 4.1). A ciascuno di essi è associata pure una struttura che sta in memoria, e che, oltre alle informazioni sullo specifico file, contiene anche il riferimento alle funzioni (i *metodi* del VFS) da usare per poterlo manipolare.

Le *dentry* “vivono” in memoria e non vengono mai salvate su disco, vengono usate per motivi di velocità, gli *inode* invece stanno su disco e vengono copiati in memoria quando serve, ed ogni cambiamento viene copiato all'indietro sul disco, gli *inode* che stanno in memoria sono *inode* del VFS ed è ad essi che puntano le singole *dentry*.

La *dcache* costituisce perciò una sorta di vista completa di tutto l'albero dei file, ovviamente per non riempire tutta la memoria questa vista è parziale (la *dcache* cioè contiene solo le *dentry* per i file per i quali è stato richiesto l'accesso), quando si vuole risolvere un nuovo *pathname* il VFS deve creare una nuova *dentry* e caricare l'*inode* corrispondente in memoria.

Questo procedimento viene eseguito dal metodo `lookup()` dell'*inode* della directory che contiene il file; questo viene installato nelle relative strutture in memoria quando si effettua il montaggio lo specifico filesystem su cui l'*inode* va a vivere.

Una volta che il VFS ha a disposizione la *dentry* (ed il relativo *inode*) diventa possibile accedere alle varie operazioni sul file come la `open` per aprire il file o la `stat` per leggere i dati dell'*inode* e passarli in user space.

L'apertura di un file richiede comunque un'altra operazione, l'allocazione di una struttura di tipo `file` in cui viene inserito un puntatore alla *dentry* e una struttura `f_ops` che contiene i puntatori ai metodi che implementano le operazioni disponibili sul file. In questo modo i processi in user space possono accedere alle operazioni attraverso detti metodi, che saranno diversi a seconda del tipo di file (o dispositivo) aperto (su questo torneremo in dettaglio in sez. 6.1.1). Un elenco delle operazioni previste dal kernel è riportato in tab. 4.2.

In questo modo per ciascun file diventano possibili una serie di operazioni (non è detto che tutte siano disponibili), che costituiscono l'interfaccia astratta del VFS. Qualora se ne voglia eseguire una, il kernel andrà ad utilizzare l'opportuna routine dichiarata in `f_ops` appropriata al tipo di file in questione.

Pertanto è possibile scrivere allo stesso modo sulla porta seriale come su un normale file di dati; ovviamente certe operazioni (nel caso della seriale ad esempio la `seek`) non saranno disponibili, però con questo sistema l'utilizzo di diversi filesystem (come quelli usati da Windows o MacOs) è immediato e (relativamente) trasparente per l'utente ed il programmatore.

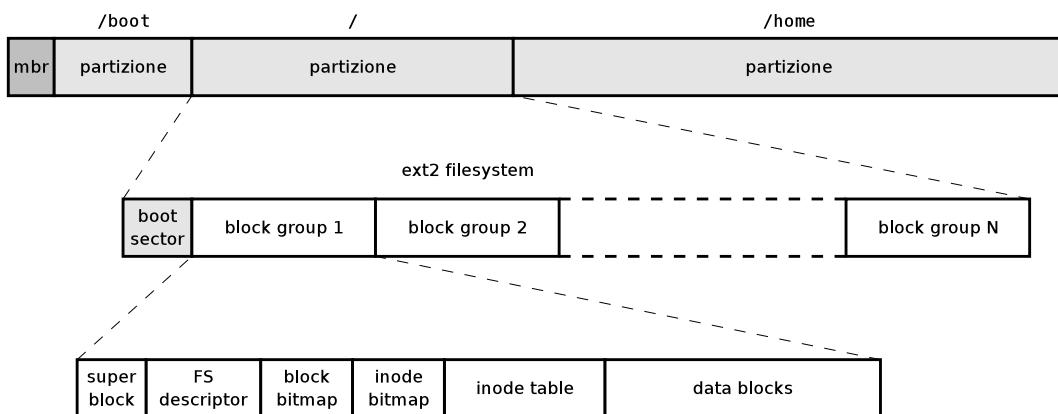
Funzione	Operazione
<code>open</code>	apre il file (vedi sez. 6.2.1).
<code>read</code>	legge dal file (vedi sez. 6.2.4).
<code>write</code>	scrive sul file (vedi sez. 6.2.5).
<code>lseek</code>	sposta la posizione corrente sul file (vedi sez. 6.2.3).
<code>ioctl</code>	accede alle operazioni di controllo (vedi sez. 6.3.6).
<code>readdir</code>	legge il contenuto di una directory
<code>poll</code>	usata nell'I/O multiplexing (vedi sez. 11.1).
<code>mmap</code>	mappa il file in memoria (vedi sez. 11.3.2).
<code>release</code>	chiamata quando l'ultimo riferimento a un file aperto è chiuso.
<code>fsync</code>	sincronizza il contenuto del file (vedi sez. 6.3.3).
<code>fasync</code>	abilita l'I/O asincrono (vedi sez. 11.2.2) sul file.

*Tabella 4.2:* Operazioni sui file definite nel VFS.

### 4.2.3 Il funzionamento di un filesystem Unix

Come già accennato in sez. 4.1.1 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 ha una sua particolare struttura e funzionalità proprie. Per questo, per il momento 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 sistema unix-like.

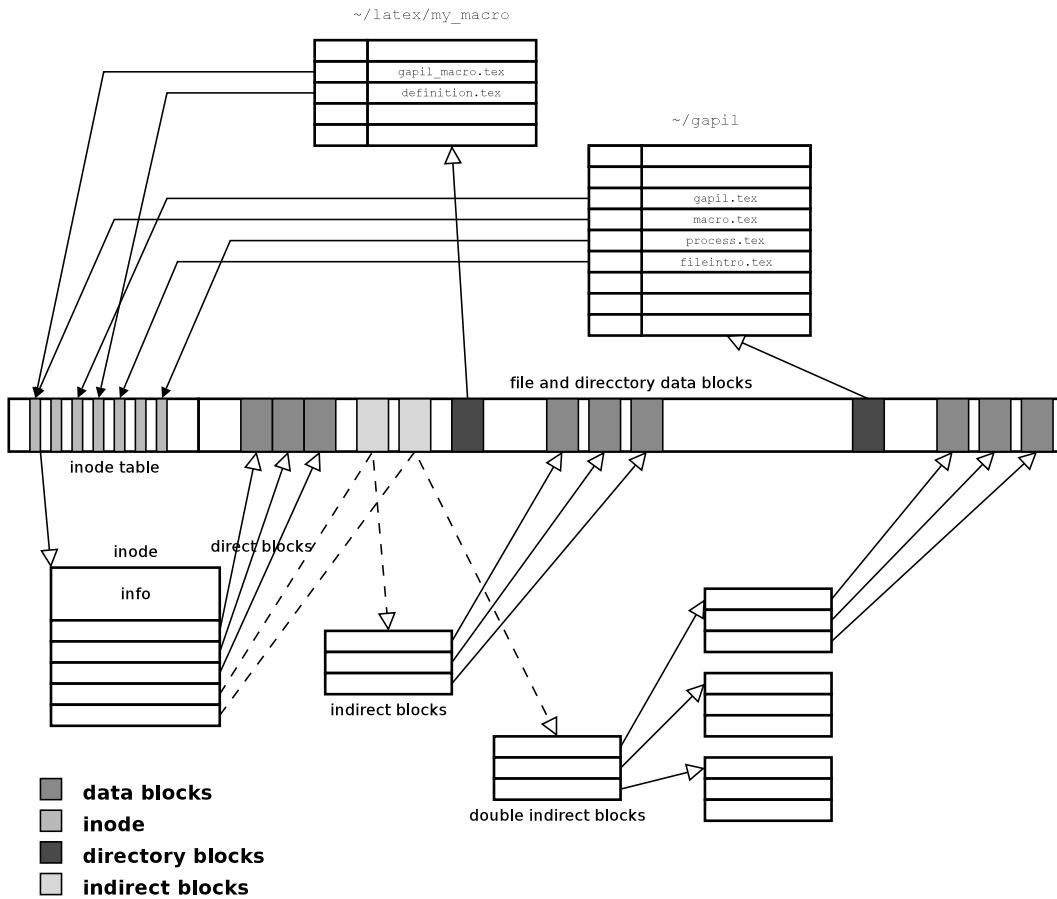
Lo spazio fisico di un disco viene usualmente diviso in partizioni; ogni partizione può contenere un filesystem. La strutturazione tipica dell'informazione su un disco è riportata in fig. 4.2; in essa si fa riferimento alla struttura del filesystem ext2, che prevede una separazione dei dati in *block group* che replicano il superblock (ma sulle caratteristiche di ext2 torneremo in sez. 4.2.4). È comunque caratteristica comune di tutti i filesystem per Unix, indipendentemente da come poi viene strutturata nei dettagli questa informazione, prevedere una divisione fra la lista degli inode e lo spazio a disposizione per i dati e le directory.



*Figura 4.2:* 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 singolo filesystem (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. 4.3.

Da fig. 4.3 si evidenziano alcune delle caratteristiche di base di un filesystem, sulle quali è bene porre attenzione visto che sono fondamentali per capire il funzionamento delle funzioni che



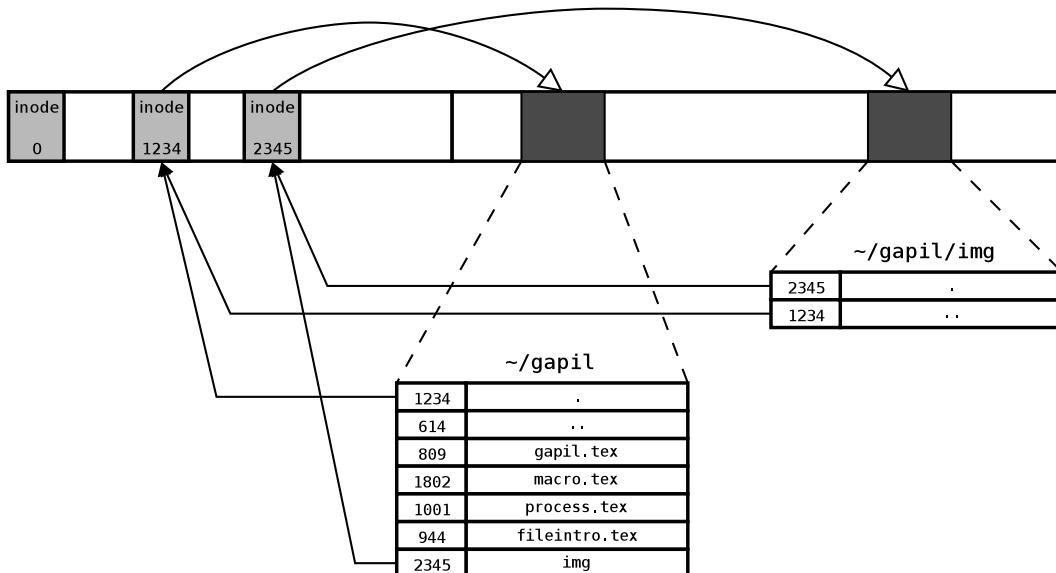
**Figura 4.3:** Strutturazione dei dati all'interno di un filesystem.

manipolano i file e le directory che tratteremo nel prossimo capitolo; in particolare è opportuno ricordare sempre che:

1. L'`inode` contiene tutte le informazioni riguardanti il file: 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` fornisce provengono dall'`inode`; dentro una directory si troverà solo il nome del file e il numero dell'`inode` ad esso associato, cioè quella che da qui in poi chiameremo una *voce* (come traduzione dell'inglese *directory entry*, che non useremo anche per evitare confusione con le *dentry* del kernel di cui si parlava in sez. 4.2.1).
2. Come mostrato in fig. 4.3 si possono avere più voci che puntano allo stesso `inode`. Ogni `inode` ha un contatore che contiene il numero di riferimenti (*link count*) che sono stati fatti ad esso; solo quando questo contatore si annulla i dati del file vengono effettivamente rimossi dal disco. Per questo la funzione per cancellare un file si chiama `unlink`, 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. Il numero di `inode` nella voce si riferisce ad un `inode` nello stesso filesystem e non ci può essere una directory che contiene riferimenti ad `inode` relativi ad altri filesystem. Questo limita l'uso del comando `ln` (che crea una nuova voce per un file esistente, con la funzione `link`) al 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 vecchia (questa è la modalità in cui opera normalmente il comando `mv` attraverso la funzione `rename`).

Infine è bene avere presente che, essendo file pure loro, esiste un numero di riferimenti anche per le directory; per cui, se a partire dalla situazione mostrata in fig. 4.3 creiamo una nuova directory `img` nella directory `gapil`, avremo una situazione come quella in fig. 4.4, dove per chiarezza abbiamo aggiunto dei numeri di inode.



**Figura 4.4:** Organizzazione dei link per le directory.

La nuova directory avrà allora 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 `.` che è sempre inserita in ogni directory; questo vale 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`.

#### 4.2.4 Il filesystem ext2

Il filesystem standard usato da Linux è il cosiddetto *second extended filesystem*, identificato dalla sigla `ext2`. Esso supporta tutte le caratteristiche di un filesystem standard Unix, è in grado di gestire nomi di file lunghi (256 caratteri, estensibili a 1012) con una dimensione massima di 4 Tb.

Oltre alle caratteristiche standard, `ext2` fornisce alcune estensioni che non sono presenti sugli altri filesystem Unix. Le principali sono le seguenti:

- i *file attributes* consentono di modificare il comportamento del kernel quando agisce su gruppi di file. Possono essere impostati su file e directory e in quest'ultimo caso i nuovi file creati nella directory ereditano i suoi attributi.
- 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 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. 5.3), 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 link 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 file immutabili (che possono solo essere letti) per la protezione di file di configurazione sensibili, o file *append-only* che possono essere aperti in scrittura solo per aggiungere dati (caratteristica utilizzabile 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. 4.3, in cui la partizione è divisa in gruppi di blocchi.<sup>8</sup>

Ciascun gruppo di blocchi contiene una copia delle informazioni essenziali del filesystem (superblock e descrittore del filesystem sono quindi ridondanti) per una maggiore affidabilità e possibilità di recupero in caso di corruzione del superblock principale.

node Number	Entry Size	Filename Size	Filename
1234	12	1	.
614	12	2	..
809	20	9	gapil.tex
1802	20	9	macro.tex
1001	20	11	process.tex
944	24	13	fileintro.tex
2345	12	3	img

**Figura 4.5:** Struttura delle directory nel *second extented filesystem*.

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.

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. 4.5; in questo modo è possibile implementare nomi per i file anche molto lunghi (fino a 1024 caratteri) senza sprecare spazio disco.

---

<sup>8</sup>non si confonda la questa definizione con quella riportata in fig. 5.2; in quel caso si fa riferimento alla struttura usata in user space per riportare i dati contenuti in una directory generica, questa fa riferimento alla struttura usata dal kernel per un filesystem ext2, definita nel file `ext2_fs.h` nella directory `include/linux` dei sorgenti del kernel.

# Capitolo 5

## File e directory

In questo capitolo tratteremo in dettaglio le modalità con cui si gestiscono file e directory, iniziando dalle funzioni di libreria che si usano per copiarli, spostarli e cambiarne i nomi. Esamineremo poi l'interfaccia che permette la manipolazione dei vari attributi di file e directory ed alla fine faremo una trattazione dettagliata su come è strutturato il sistema base di protezioni e controllo dell'accesso ai file e sulle funzioni che ne permettono la gestione. Tutto quello che riguarda invece la manipolazione del contenuto dei file è lasciato ai capitoli successivi.

### 5.1 La gestione di file e directory

Come già accennato in sez. 4.2.3 in un sistema unix-like la gestione dei file ha delle caratteristiche specifiche che derivano direttamente dall'architettura del sistema.

In questa sezione esamineremo le funzioni usate per la manipolazione di file e directory, per la creazione di link simbolici e diretti, per la gestione e la lettura delle directory.

In particolare ci soffermeremo sulle conseguenze che derivano dall'architettura dei filesystem illustrata nel capitolo precedente per quanto riguarda il comportamento delle varie funzioni.

#### 5.1.1 Le funzioni link e unlink

Una caratteristica comune a diversi sistemi operativi è quella di poter creare dei nomi fintizi (come gli alias del 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 tali collegamenti sono usualmente chiamati *link*; ma data l'architettura del sistema riguardo la gestione dei file (ed in particolare quanto trattato in sez. 4.2) ci sono due metodi sostanzialmente diversi per fare questa operazione.

Come spiegato in sez. 4.2.3 l'accesso al contenuto di un file su disco avviene passando attraverso il suo inode, che è la struttura usata dal kernel che lo identifica univocamente all'interno di un singolo filesystem. Il nome del file che si trova nella voce di una directory è solo un'etichetta, mantenuta all'interno della directory, che viene associata ad un puntatore che fa riferimento al suddetto inode.

Questo significa che, fintanto che si resta sullo stesso filesystem, la realizzazione di un link è immediata, ed uno stesso file può avere tanti nomi diversi, dati da altrettante diverse associazioni allo stesso inode di etichette diverse in directory diverse. Si noti anche che nessuno di questi nomi viene ad assumere una particolare preferenza o originalità rispetto agli altri, in quanto tutti fanno comunque riferimento allo stesso inode.

Per aggiungere ad una directory una voce che faccia riferimento ad un inode già esistente si

utilizza la funzione `link`; si suole chiamare questo tipo di associazione un collegamento diretto (o *hard link*). Il prototipo della funzione è:

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath)
    Crea un nuovo collegamento diretto.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore nel qual caso `errno` viene impostata ai valori:

<b>EXDEV</b>	<code>oldpath</code> e <code>newpath</code> non sono sullo stesso filesystem.
<b>EPERM</b>	il filesystem che contiene <code>oldpath</code> e <code>newpath</code> non supporta i link diretti o è una directory.
<b>EEXIST</b>	un file (o una directory) con quel nome esiste di già.
<b>EMLINK</b>	ci sono troppi link al file <code>oldpath</code> (il numero massimo è specificato dalla variabile <code>LINK_MAX</code> , vedi sez. 8.1.1).
ed inoltre <code>EACCES</code> , <code>ENAMETOOLONG</code> , <code>ENOTDIR</code> , <code>EFAULT</code> , <code>ENOMEM</code> , <code>EROFS</code> , <code>ELOOP</code> , <code>ENOSPC</code> , <code>EIO</code> .	

La funzione crea sul *pathname* `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 una voce nella directory specificata da `newpath` e ad aumentare di uno il numero di riferimenti al file (riportato nel campo `st_nlink` della struttura `stat`, vedi sez. 5.2.1) aggiungendo il nuovo nome ai precedenti. Si noti che uno stesso file può essere così chiamato con vari nomi in diverse directory.

Per quanto dicevamo in sez. 4.2.3 la creazione di un collegamento diretto è possibile solo se entrambi i *pathname* sono nello stesso filesystem; inoltre il filesystem deve supportare i collegamenti diretti (il meccanismo non è disponibile ad esempio con il filesystem *vfat* di Windows).

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 l'esempio mostrato in sez. 5.1.3, dove riprenderemo il discorso) che molti programmi non sono in grado di gestire e la cui rimozione diventerebbe estremamente complicata (in genere per questo tipo di errori occorre far girare il programma `fsck` per riparare il filesystem).

Data la pericolosità di questa operazione e la disponibilità dei link simbolici che possono fornire la stessa funzionalità senza questi problemi, nei filesystem usati in Linux questa caratteristica è stata completamente disabilitata, e al tentativo di creare un link diretto ad una directory la funzione restituisce l'errore `EPERM`.

La rimozione di un file (o più precisamente della voce che lo riferenzia all'interno di una directory) si effettua con la funzione `unlink`; il suo prototipo è il seguente:

```
#include <unistd.h>
int unlink(const char *pathname)
    Cancella un file.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso il file non viene toccato. La variabile `errno` viene impostata secondo i seguenti codici di errore:

<b>EISDIR</b>	<code>pathname</code> si riferisce ad una directory. <sup>1</sup>
<b>EROFS</b>	<code>pathname</code> è su un filesystem montato in sola lettura.
<b>EISDIR</b>	<code>pathname</code> fa riferimento a una directory.
ed inoltre: <code>EACCES</code> , <code>EFAULT</code> , <code>ENOENT</code> , <code>ENOTDIR</code> , <code>ENOMEM</code> , <code>EROFS</code> , <code>ELOOP</code> , <code>EIO</code> .	

<sup>1</sup>questo è un valore specifico ritornato da Linux che non consente l'uso di `unlink` con le directory (vedi sez. 5.1.2). Non è 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.

La funzione cancella il nome specificato da `pathname` nella relativa directory e decrementa il numero di riferimenti nel relativo inode. Nel caso di link simbolico cancella il link simbolico; nel caso di socket, fifo o file di dispositivo rimuove il nome, ma come per i file i processi che hanno aperto uno di questi oggetti possono continuare ad utilizzarlo.

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 sulla directory che la contiene (affronteremo in dettaglio l'argomento dei permessi di file e directory in sez. 5.3). Se inoltre lo *sticky* bit (vedi sez. 5.3.3) è impostato occorrerà anche essere proprietari del file o proprietari della directory (o root, per cui nessuna delle restrizioni è applicata).

Una delle caratteristiche di queste funzioni è che la creazione/rimozione del nome dalla directory e l'incremento/decremento del numero di riferimenti nell'inode devono essere effettuati in maniera atomica (si veda sez. 3.5.1) senza possibili interruzioni fra le due operazioni. Per questo entrambe queste funzioni sono realizzate tramite una singola system call.

Si ricordi infine che un file non viene eliminato dal disco fintanto che tutti i riferimenti ad esso sono stati cancellati: solo quando il *link count* mantenuto nell'inode diventa zero lo spazio occupato su disco viene rimosso (si ricordi comunque che a questo si aggiunge sempre un'ulteriore condizione,<sup>2</sup> e cioè che non ci siano processi che abbiano il suddetto file aperto).

Questa proprietà viene spesso usata per essere sicuri di non lasciare file temporanei su disco in caso di crash dei programmi; la tecnica è quella di aprire il file e chiamare `unlink` subito dopo, in questo modo il contenuto del file è sempre disponibile all'interno del processo attraverso il suo file descriptor (vedi sez. 6.1.1) fintanto che il processo non chiude il file, ma non ne resta traccia in nessuna directory, e lo spazio occupato su disco viene immediatamente rilasciato alla conclusione del processo (quando tutti i file vengono chiusi).

### 5.1.2 Le funzioni `remove` e `rename`

Al contrario di quanto avviene con altri Unix, in Linux non è possibile usare `unlink` sulle directory; per cancellare una directory si può usare la funzione `rmdir` (vedi sez. 5.1.4), oppure la funzione `remove`.

Questa è la funzione prevista dallo standard ANSI C per cancellare un file o una directory (e funziona anche per i sistemi che non supportano i link diretti). Per i file è identica a `unlink` e per le directory è identica a `rmdir`; il suo prototipo è:

```
#include <stdio.h>
int remove(const char *pathname)
    Cancella un nome dal filesystem.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso il file non viene toccato.

I codici di errore riportati in `errno` sono quelli della chiamata utilizzata, pertanto si può fare riferimento a quanto illustrato nelle descrizioni di `unlink` e `rmdir`.

La funzione utilizza la funzione `unlink`<sup>3</sup> per cancellare i file e la funzione `rmdir` per cancellare le directory; si tenga presente che per alcune implementazioni del protocollo NFS utilizzare questa funzione può comportare la scomparsa di file ancora in uso.

<sup>2</sup>come vedremo in cap. 6 il kernel mantiene anche una tabella dei file aperti nei vari processi, 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.

<sup>3</sup>questo vale usando le *glibc*; nelle libc4 e nelle libc5 la funzione `remove` è un semplice alias alla funzione `unlink` e quindi non può essere usata per le directory.

Per cambiare nome ad un file o a una directory (che devono comunque essere nello stesso filesystem) si usa invece la funzione `rename`,<sup>4</sup> il cui prototipo è:

```
#include <stdio.h>
int rename(const char *oldpath, const char *newpath)
    Rinomina un file.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso il file non viene toccato. La variabile `errno` viene impostata secondo i seguenti codici di errore:

<b>EISDIR</b>	<code>newpath</code> è una directory mentre <code>oldpath</code> non è una directory.
<b>EXDEV</b>	<code>oldpath</code> e <code>newpath</code> non sono sullo stesso filesystem.
<b>ENOTEMPTY</b>	<code>newpath</code> è una directory già esistente e non vuota.
<b>EBUSY</b>	o <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 mount point).
<b>EINVAL</b>	<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 se stessa.
<b>ENOTDIR</b>	Uno dei componenti dei <i>pathname</i> non è una directory o <code>oldpath</code> è una directory e <code>newpath</code> esiste e non è una directory.

ed inoltre `EACCES`, `EPERM`, `EMLINK`, `ENOENT`, `ENOMEM`, `EROFS`, `ELOOP` e `ENOSPC`.

La funzione rinomina il file `oldpath` in `newpath`, eseguendo se necessario lo spostamento di un file fra directory diverse. Eventuali altri link diretti allo stesso file non vengono influenzati.

Il comportamento della funzione è diverso a seconda che si voglia rinominare un file o una directory; se ci riferisce a un file allora `newpath`, se esiste, non deve essere una directory (altrimenti si ha l'errore `EISDIR`). Nel caso `newpath` indichi un file esistente questo viene cancellato e rimpiazzato (atomicamente).

Se `oldpath` è una directory allora `newpath`, se esiste, deve essere una directory vuota, altrimenti si avranno gli errori `ENOTDIR` (se non è una directory) o `ENOTEMPTY` (se non è vuota). Chiaramente `newpath` non può contenere `oldpath` altrimenti si avrà un errore `EINVAL`.

Se `oldpath` si riferisce a un link simbolico questo sarà rinominato; se `newpath` è un link simbolico verrà cancellato come qualunque altro file. Infine qualora `oldpath` e `newpath` siano due nomi dello stesso file lo standard POSIX prevede che la funzione non dia errore, e non faccia nulla, lasciando entrambi i nomi; Linux segue questo standard, anche se, come fatto notare dal manuale delle *glibc*, il comportamento più ragionevole sarebbe quello di cancellare `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 può esistere cioè nessun istante in cui un altro processo può trovare attivi entrambi i nomi dello stesso file, o, in caso di sostituzione di un file esistente, non trovare quest'ultimo prima che la sostituzione sia stata eseguita.

In ogni caso se `newpath` esiste e l'operazione fallisce per un qualche motivo (come un crash del kernel), `rename` garantisce di lasciare presente un'istanza di `newpath`. Tuttavia nella sovrascrittura potrà esistere una finestra in cui sia `oldpath` che `newpath` fanno riferimento allo stesso file.

### 5.1.3 I link simbolici

Come abbiamo visto in sez. 5.1.1 la funzione `link` crea riferimenti agli inode, pertanto 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 link diretto ad una directory.

Per ovviare a queste limitazioni i sistemi Unix supportano un'altra forma di link (i cosiddetti *soft link* o *symbolic link*), che sono, come avviene in altri sistemi operativi, dei file speciali che

<sup>4</sup>la funzione è definita dallo standard ANSI C, ma si applica solo per i file, lo standard POSIX estende la funzione anche alle directory.

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 link diretti, a delle directory, ed anche a file che non esistono ancora.

Il sistema funziona in quanto i link simbolici sono riconosciuti come tali dal kernel<sup>5</sup> per cui alcune funzioni di libreria (come `open` o `stat`) quando ricevono come argomento un link simbolico vengono automaticamente applicate al file da esso specificato. La funzione che permette di creare un nuovo link simbolico è `symlink`, ed il suo prototipo è:

```
#include <unistd.h>
int symlink(const char *oldpath, const char *newpath)
    Crea un nuovo link simbolico di nome newpath il cui contenuto è oldpath.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso la variabile `errno` assumerà i valori:

<code>EPERM</code>	il filesystem che contiene <code>newpath</code> non supporta i link simbolici.
<code>ENOENT</code>	una componente di <code>newpath</code> non esiste o <code>oldpath</code> è una stringa vuota.
<code>EEXIST</code>	esiste già un file <code>newpath</code> .
<code>EROFS</code>	<code>newpath</code> è su un filesystem montato in sola lettura.

ed inoltre `EFAULT`, `EACCES`, `ENAMETOOLONG`, `ENOTDIR`, `ENOMEM`, `ELOOP`, `ENOSPC` e `EIO`.

Si tenga presente che la funzione non effettua nessun controllo sull'esistenza di un file di nome `oldpath`, ma si limita ad inserire quella stringa nel link simbolico. Pertanto un link simbolico può anche riferirsi ad un file che non esiste: in questo caso si ha quello che viene chiamato un *dangling link*, letteralmente un *link ciondolante*.

Come accennato i link simbolici sono risolti automaticamente dal kernel all'invocazione delle varie system call; in tab. 5.1 si è riportato un elenco dei comportamenti delle varie funzioni di libreria che operano sui file nei confronti della risoluzione dei link simbolici, specificando quali seguono il link simbolico e quali invece possono operare direttamente sul suo contenuto.

Funzione	Segue il link	Non segue il link
<code>access</code>	•	—
<code>chdir</code>	•	—
<code>chmod</code>	•	—
<code>chown</code>	—	•
<code>creat</code>	•	—
<code>exec</code>	•	—
<code>lchown</code>	•	•
<code>link</code>	—	—
<code>lstat</code>	—	•
<code>mkdir</code>	•	—
<code>mkfifo</code>	•	—
<code>mknod</code>	•	—
<code>open</code>	•	—
<code>opendir</code>	•	—
<code>pathconf</code>	•	—
<code>readlink</code>	—	•
<code>remove</code>	—	•
<code>rename</code>	—	•
<code>stat</code>	•	—
<code>truncate</code>	•	—
<code>unlink</code>	—	•

**Tabella 5.1:** Uso dei link simbolici da parte di alcune funzioni.

Si noti che non si è specificato il comportamento delle funzioni che operano con i file de-

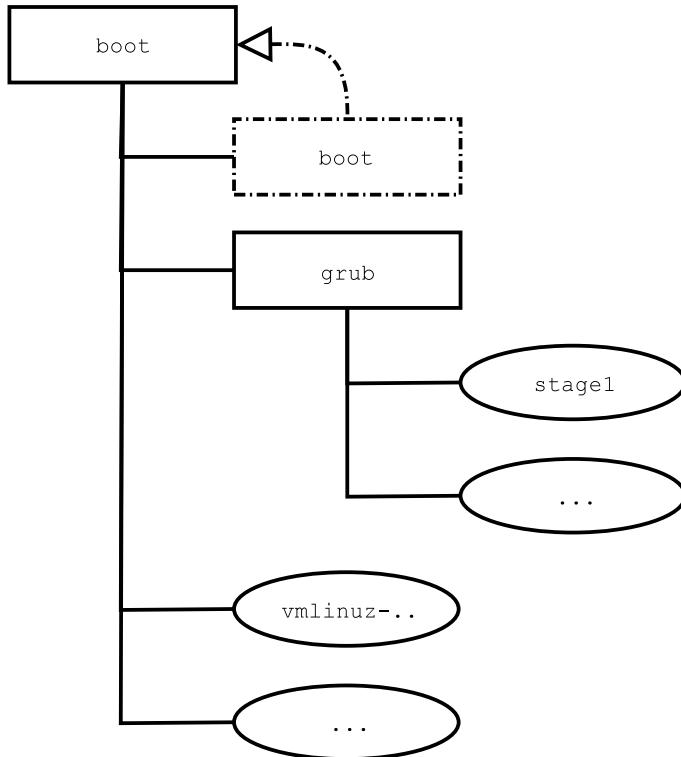
<sup>5</sup>è uno dei diversi tipi di file visti in tab. 4.1, contrassegnato come tale nell'inode, e riconoscibile dal valore del campo `st_mode` della struttura `stat` (vedi sez. 5.2.1).

scriptor, in quanto la risoluzione del link simbolico viene in genere effettuata dalla funzione che restituisce il file descriptor (normalmente la `open`, vedi sez. 6.2.1) e tutte le operazioni seguenti fanno riferimento solo a quest'ultimo.

Dato che, come indicato in tab. 5.1, funzioni come la `open` seguono i link simbolici, occorrono funzioni apposite per accedere alle informazioni del link invece che a quelle del file a cui esso fa riferimento. Quando si vuole leggere il contenuto di un link simbolico si usa la funzione `readlink`, il cui prototipo è:

```
#include <unistd.h>
int readlink(const char *path, char *buff, size_t size)
    Legge il contenuto del link simbolico indicato da path nel buffer buff di dimensione size.
La funzione restituisce il numero di caratteri letti dentro buff o -1 per un errore, nel qual caso la variabile errno assumerà i valori:
EINVAL      path non è un link simbolico o size non è positiva.
ed inoltre ENOTDIR, ENAMETOOLONG, ENOENT, EACCES, ELOOP, EIO,EFAULT e ENOMEM.
```

La funzione apre il link simbolico, ne legge il contenuto, lo scrive nel buffer, e lo richiude. Si tenga presente che la funzione non termina la stringa con un carattere nullo e la tronca alla dimensione specificata da `size` per evitare di sovrascrivere oltre le dimensioni del buffer.



**Figura 5.1:** Esempio di loop nel filesystem creato con un link simbolico.

Un caso comune che si può avere con i link simbolici è la creazione dei cosiddetti *loop*. La situazione è illustrata in fig. 5.1, che riporta la struttura della directory `/boot`. Come si vede si è creato al suo interno un link simbolico che punta di nuovo a `/boot`.<sup>6</sup>

<sup>6</sup>il loop mostrato in fig. 5.1 è un usato per poter permettere a `grub` (un bootloader in grado di leggere direttamente da vari filesystem il file da lanciare come sistema operativo) di vedere i file contenuti nella directory `/boot` con lo stesso *pathname* con cui verrebbero visti dal sistema operativo, anche se essi si trovano, come accade spesso, su una partizione separata (che `grub`, all'avvio, vede come radice).

Questo può causare problemi per tutti quei programmi che effettuano la scansione di una directory senza tener conto dei link simbolici, ad esempio se lanciassimo un comando del tipo `grep -r linux *`, il loop nella directory porterebbe il comando 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 un numero limitato di link simbolici, il cui valore limite è specificato dalla costante `MAXSYMLINKS`. Qualora questo limite venga superato viene generato un errore ed `errno` viene impostata al valore `ELOOP`.

Un punto da tenere sempre presente è che, come abbiamo accennato, un link simbolico può fare riferimento anche ad un file che non esiste; ad esempio possiamo creare un file temporaneo nella nostra directory con un link del tipo:

```
$ ln -s /tmp/tmp_file temporaneo
```

anche se `/tmp/tmp_file` non esiste. Questo può generare confusione, in quanto aprendo in scrittura `temporaneo` verrà creato `/tmp/tmp_file` e scritto; ma accedendo in sola lettura a `temporaneo`, ad esempio con `cat`, otterremmo:

```
$ cat temporaneo
cat: temporaneo: No such file or directory
```

con un errore che può sembrare sbagliato, dato che un'ispezione con `ls` ci mostrerebbe invece l'esistenza di `temporaneo`.

#### 5.1.4 La creazione e la cancellazione delle directory

Benché in sostanza le directory non siano altro che dei file contenenti elenchi di nomi ed inode, non è possibile trattarle come file ordinari e devono essere create direttamente dal kernel attraverso una opportuna system call.<sup>7</sup> La funzione usata per creare una directory è `mkdir`, ed il suo prototipo è:

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *dirname, mode_t mode)
    Crea una nuova directory.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<code>EEXIST</code>	Un file (o una directory) con quel nome esiste di già.
<code>EACCES</code>	Non c'è il permesso di scrittura per la directory in cui si vuole inserire la nuova directory.
<code>EMLINK</code>	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.
<code>ENOSPC</code>	Non c'è abbastanza spazio sul file system per creare la nuova directory o si è esaurita la quota disco dell'utente.

ed inoltre anche `EPERM`, `EFAULT`, `ENAMETOOLONG`, `ENOENT`, `ENOTDIR`, `ENOMEM`, `ELOOP`, `EROFS`.

La funzione crea una nuova directory vuota, che contiene cioè solo le due voci standard (`.` e `..`), con il nome indicato dall'argomento `dirname`. Il nome può essere indicato sia come *pathname* assoluto che relativo.

---

<sup>7</sup>questo permette anche, attraverso l'uso del VFS, l'utilizzo di diversi formati per la gestione dei suddetti elenchi.

I permessi di accesso alla directory (vedi sez. 5.3) sono specificati da `mode`, i cui possibili valori sono riportati in tab. 5.9; questi sono modificati dalla maschera di creazione dei file (si veda sez. 5.3.7). La titolarità della nuova directory è impostata secondo quanto riportato in sez. 5.3.4.

La funzione per la cancellazione di una directory è `rmdir`, il suo prototipo è:

```
#include <sys/stat.h>
int rmdir(const char *dirname)
    Cancella una directory.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<b>EPERM</b>	Il filesystem non supporta la cancellazione di directory, oppure la directory che contiene <code>dirname</code> ha lo sticky bit impostato e l'user-ID effettivo del processo non corrisponde al proprietario della directory.
<b>EACCES</b>	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 <code>dirname</code> .
<b>EBUSY</b>	La directory specificata è la directory di lavoro o la radice di qualche processo.
<b>ENOTEMPTY</b>	La directory non è vuota.
ed inoltre anche <b>EFAULT</b> , <b>ENAMETOOLONG</b> , <b>ENOENT</b> , <b>ENOTDIR</b> , <b>ENOMEM</b> , <b>ELOOP</b> , <b>EROFS</b> .	

La funzione cancella la directory `dirname`, che deve essere vuota (la directory deve cioè contenere soltanto le due voci standard . e ..). Il nome può essere indicato con il *pathname* assoluto o relativo.

La modalità con cui avviene la cancellazione è analoga a quella di `unlink`: fintanto che il numero di link all'inode della directory non diventa nullo e nessun processo ha la directory aperta lo spazio occupato su disco non viene rilasciato. Se un processo ha la directory aperta la funzione rimuove il link all'inode e nel caso sia l'ultimo, pure le voci standard . e .., a questo punto il kernel non consentirà di creare più nuovi file nella directory.

### 5.1.5 La creazione di file speciali

Finora abbiamo parlato esclusivamente di file, directory e link simbolici; in sez. 4.1.2 abbiamo visto però che il sistema prevede pure degli altri tipi di file speciali, come i file di dispositivo e le fifo (i socket sono un caso a parte, che vedremo in cap. 14).

La manipolazione delle caratteristiche di questi file e la loro cancellazione può essere effettuata con le stesse funzioni che operano sui file regolari; ma quando li si devono creare sono necessarie delle funzioni apposite. La prima di queste funzioni è `mknod`, il suo 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 inode, si usa per creare i file speciali.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<b>EPERM</b>	Non si hanno privilegi sufficienti a creare l'inode, o il filesystem su cui si è cercato di creare <code>pathname</code> non supporta l'operazione.
<b>EINVAL</b>	Il valore di <code>mode</code> non indica un file, una fifo o un dispositivo.
<b>EEXIST</b>	<code>pathname</code> esiste già o è un link simbolico.
ed inoltre anche <b>EFAULT</b> , <b>EACCES</b> , <b>ENAMETOOLONG</b> , <b>ENOENT</b> , <b>ENOTDIR</b> , <b>ENOMEM</b> , <b>ELOOP</b> , <b>ENOSPC</b> , <b>EROFS</b> .	

La funzione permette di creare un file speciale, ma si può usare anche per creare file regolari e fifo; l'argomento `mode` specifica il tipo di file che si vuole creare ed i relativi permessi, secondo i valori riportati in tab. 5.4, che vanno combinati con un OR binario. I permessi sono comunque modificati nella maniera usuale dal valore di `umask` (si veda sez. 5.3.7).

Per il tipo di file può essere specificato solo uno fra: `S_IFREG` per un file regolare (che sarà creato vuoto), `S_IFBLK` per un device a blocchi, `S_IFCHR` per un device a caratteri e `S_IFIFO` per una fifo. Un valore diverso comporterà l'errore `EINVAL`. Qualora si sia specificato in `mode` un file di dispositivo, il valore di `dev` viene usato per indicare a quale dispositivo si fa riferimento.

Solo l'amministratore può creare un file di dispositivo o un file regolare usando questa funzione; ma in Linux<sup>8</sup> l'uso per la creazione di una fifo è consentito anche agli utenti normali.

I nuovi inode creati con `mknod` apparterranno al proprietario e al gruppo del processo che li ha creati, a meno che non si sia attivato il bit `sgid` per la directory o sia stata attivata la semantica BSD per il filesystem (si veda sez. 5.3.4) in cui si va a creare l'inode.

Per creare una fifo (un file speciale, su cui torneremo in dettaglio in sez. 12.1.4) lo standard POSIX specifica l'uso della funzione `mkfifo`, il cui prototipo è:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode)
    Crea una fifo.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori `EACCES`, `EXIST`, `ENAMETOOLONG`, `ENOENT`, `ENOSPC`, `ENOTDIR` e `EROFS`.

La funzione crea la fifo `pathname` con i permessi `mode`. Come per `mknod` il file `pathname` non deve esistere (neanche come link simbolico); al solito i permessi specificati da `mode` vengono modificati dal valore di `umask`.

### 5.1.6 Accesso alle directory

Benché le directory alla fine non siano altro che dei file che contengono delle liste di nomi ed 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. 6.2.1 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 del kernel prevede una apposita funzione per la lettura delle directory.

Tutto questo si riflette nello standard POSIX<sup>9</sup> 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 di cap. 7). 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 restituisce un puntatore al *directory stream* in caso di successo e `NULL` per un errore, nel qual caso `errno` assumerà i valori `EACCES`, `EMFILE`, `ENFILE`, `ENOMEM` e `ENOTDIR`.

<sup>8</sup>la funzione non è prevista dallo standard POSIX, e deriva da SVr4, con appunto questa differenza e diversi codici di errore.

<sup>9</sup>le funzioni sono previste pure in BSD e SVID.

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.

Dato che le directory sono comunque dei file, in alcuni casi può servire conoscere il *file descriptor* associato ad un *directory stream*, a questo scopo si può usare la funzione **dirfd**, il cui prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
int dirfd(DIR * dir)
    Restituisce il file descriptor associato ad un directory stream.
```

La funzione restituisce il file descriptor (un valore positivo) in caso di successo e -1 in caso di errore.

La funzione<sup>10</sup> restituisce il file descriptor associato al *directory stream* **dir**, essa è disponibile solo definendo **\_BSD\_SOURCE** o **\_SVID\_SOURCE**. Di solito si utilizza questa funzione in abbinamento alla funzione **fchdir** per cambiare la directory di lavoro (vedi sez. 5.1.7) a quella relativa allo stream che si sta esaminando.

La lettura di una voce della directory viene effettuata attraverso la funzione **readdir**; il suo prototipo è:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dir)
    Legge una voce dal directory stream.
```

La funzione restituisce il puntatore alla struttura contenente i dati in caso di successo e **NULL** altrimenti, in caso di descrittore non valido **errno** assumerà il valore **EBADF**, il valore **NULL** viene restituito anche quando si raggiunge la fine dello stream.

La funzione legge la voce corrente nella directory, posizionandosi sulla voce successiva. I dati vengono memorizzati in una struttura **dirent** (la cui definizione<sup>11</sup> è riportata in fig. 5.2). La funzione restituisce il puntatore alla struttura; si tenga presente però che quest'ultima è allocata staticamente, per cui viene sovrascritta tutte le volte che si ripete la lettura di una voce sullo stesso stream.

Di questa funzione esiste anche una versione rientrante, **readdir\_r**, 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 restituisce 0 in caso di successo e -1 in caso di errore, gli errori sono gli stessi di **readdir**.

La funzione restituisce in **result** (come *value result argument*) l'indirizzo dove sono stati salvati i dati, che di norma corrisponde a quello della struttura precedentemente allocata e specificata dall'argomento **entry** (anche se non è assicurato che la funzione usi lo spazio fornito dall'utente).

<sup>10</sup>questa funzione è una estensione di BSD non presente in POSIX, introdotta con BSD 4.3-Reno; è presente in Linux con le libc5 (a partire dalla versione 5.1.2) e con le *glibc*.

<sup>11</sup>la definizione è quella usata a 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 (ed infatti la macro **\_DIRENT\_HAVE\_D\_NAMLEN** non è definita).

I vari campi di `dirent` contengono le informazioni relative alle voci presenti nella directory; sia BSD che SVr4<sup>12</sup> prevedono che siano sempre presenti il campo `d_name`, che contiene il nome del file nella forma di una stringa terminata da uno zero,<sup>13</sup> ed il campo `d_ino`, che contiene il numero di inode cui il file è associato (di solito corrisponde al campo `st_ino` di `stat`).

---

```
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 */
    char d_name[256];                 /* We must not include limits.h! */
};
```

---

*Figura 5.2:* La struttura `dirent` per la lettura delle informazioni dei file.

La presenza di ulteriori campi opzionali è segnalata dalla definizione di altrettante macro nella forma `_DIRENT_HAVE_D_XXX` dove `XXX` è il nome del relativo campo; nel nostro caso sono definite le macro `_DIRENT_HAVE_D_TYPE`, `_DIRENT_HAVE_D_OFF` e `_DIRENT_HAVE_D_RECLEN`.

Valore	Significato
<code>DT_UNKNOWN</code>	tipo sconosciuto.
<code>DT_REG</code>	file normale.
<code>DT_DIR</code>	directory.
<code>DT_FIFO</code>	fifo.
<code>DT_SOCK</code>	socket.
<code>DT_CHR</code>	dispositivo a caratteri.
<code>DT_BLK</code>	dispositivo a blocchi.

*Tabella 5.2:* Costanti che indicano i vari tipi di file nel campo `d_type` della struttura `dirent`.

Per quanto riguarda il significato dei campi opzionali, il campo `d_type` indica il tipo di file (fifo, directory, link simbolico, ecc.); i suoi possibili valori<sup>14</sup> sono riportati in tab. 5.2; per la conversione da e verso l'analogo valore mantenuto dentro il campo `st_mode` di `stat` sono definite anche due macro di conversione `IFTODT` e `DTTOIF`:

```
int IFTODT(mode_t MODE)
    Converte il tipo di file dal formato di st_mode a quello di d_type.
mode_t DTTOIF(int DTTYPE)
    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`,<sup>15</sup> il cui prototipo è:

```
#include <dirent.h>
void seekdir(DIR *dir, off_t offset)
    Cambia la posizione all'interno di un directory stream.
```

<sup>12</sup>POSIX prevede invece solo la presenza del campo `d_fileno`, identico `d_ino`, che in Linux è definito come alias di quest'ultimo. Il campo `d_name` è considerato dipendente dall'implementazione.

<sup>13</sup>lo standard POSIX non specifica una lunghezza, ma solo un limite `NAME_MAX`; in SVr4 la lunghezza del campo è definita come `NAME_MAX+1` che di norma porta al valore di 256 byte usato anche in Linux.

<sup>14</sup>fino alla versione 2.1 delle *glibc* questo campo, pur presente nella struttura, non è implementato, e resta sempre al valore `DT_UNKNOWN`.

<sup>15</sup>sia questa funzione che `telldir`, sono estensioni prese da BSD, non previste dallo standard POSIX.

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 `telldir`, che legge la posizione corrente; il prototipo di quest'ultima è:

```
#include <dirent.h>
off_t telldir(DIR *dir)
    Ritorna la posizione corrente in un directory stream.
```

La funzione restituisce la posizione corrente nello stream (un numero positivo) in caso di successo, e -1 altrimenti, nel qual caso `errno` assume solo il valore di `EBADF`, corrispondente ad un valore errato per `dir`.

La sola funzione di posizionamento nello stream prevista 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.
```

Una volta completate le operazioni si può chiudere il *directory stream* 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 restituisce 0 in caso di successo e -1 altrimenti, nel qual caso `errno` assume il valore `EBADF`.

A parte queste funzioni di base in BSD 4.3 è stata 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`<sup>16</sup> ed il suo prototipo è:

```
#include <dirent.h>
int scandir(const char *dir, struct dirent ***namelist, int(*select)(const struct
    dirent *), int(*compar)(const struct dirent **, const struct dirent **))
    Esegue una scansione di un directory stream.
```

La funzione restituisce in caso di successo il numero di voci trovate, e -1 altrimenti.

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 (`select`) e l'ordinamento di tutte le voci selezionate (`compar`).

La funzione legge tutte le voci della directory indicata dall'argomento `dir`, passando ciascuna di esse come argomento alla funzione di `select`; se questa ritorna un valore diverso da zero la voce viene inserita in una struttura allocata dinamicamente con `malloc`, qualora si specifichi un valore `NULL` per `select` vengono selezionate tutte le voci. Tutte le voci selezionate vengono poi inserite in una lista (anch'essa allocata con `malloc`, che viene riordinata tramite `qsort` usando la funzione `compar` come criterio di ordinamento; alla fine l'indirizzo della lista ordinata è restituito nell'argomento `namelist`.

Per l'ordinamento sono disponibili anche due funzioni predefinite, `alphasort` e `versionsort`, i cui prototipi sono:

---

<sup>16</sup>in Linux questa funzione è stata introdotta fin dalle libc4.

```
#include <dirent.h>
int alphasort(const void *a, const void *b)
int versionsort(const void *a, const void *b)
    Funzioni per l'ordinamento delle 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.

La funzione `alphasort` deriva da BSD ed è presente in Linux fin dalle libc4<sup>17</sup> e deve essere specificata come argomento `compare` per ottenere un ordinamento alfabetico (secondo il valore del campo `d_name` delle varie voci). Le glibc prevedono come estensione<sup>18</sup> 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. 5.3, dove si è riportata la sezione principale di un programma che, usando la routine di scansione illustrata in fig. 5.4, stampa i nomi dei file contenuti in una directory e la relativa dimensione (in sostanza una versione semplificata del comando `ls`).

---

```

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 /* computation function for DirScan */
8 int do_ls(struct dirent * direntry);
9 /* main body */
10 int main(int argc, char *argv[])
11 {
12     ...
13     if ((argc - optind) != 1)           /* There must be remaing parameters */
14         printf("Wrong number of arguments %d\n", argc - optind);
15         usage();
16     }
17     DirScan(argv[1], do_ls);
18     exit(0);
19 }
20 /*
21 * Routine to print file name and size inside DirScan
22 */
23 int do_ls(struct dirent * direntry)
24 {
25     struct stat data;
26
27     stat(direntry->d_name, &data);
28     /* get stat data */
29     printf("File: %s \t size: %d\n", direntry->d_name, data.st_size);
30 }
```

---

**Figura 5.3:** Esempio di codice per eseguire la lista dei file contenuti in una directory.

Il programma è estremamente semplice; in fig. 5.3 si è omessa la parte di gestione delle

<sup>17</sup>la versione delle libc4 e libc5 usa però come argomenti dei puntatori a delle strutture `dirent`; le glibc usano il prototipo originario di BSD, mostrato anche nella definizione, che prevede puntatori a `void`.

<sup>18</sup>le glibc, a partire dalla versione 2.1, effettuano anche l'ordinamento alfabetico tenendo conto delle varie localizzazioni, usando `strcoll` al posto di `strcmp`.

opzioni (che prevede solo l'uso di una funzione per la stampa della sintassi, anch'essa omessa) ma il codice completo potrà essere trovato coi sorgenti allegati nel file `myls.c`.

In sostanza tutto quello che fa il programma, dopo aver controllato (10-13) di avere almeno un parametro (che indicherà la directory da esaminare) è chiamare (14) la funzione `DirScan` per eseguire la scansione, usando la funzione `do_ls` (20-26) per fare tutto il lavoro.

Quest'ultima si limita (23) a chiamare `stat` sul file indicato dalla directory entry passata come argomento (il cui nome è appunto `direntry->d_name`), memorizzando in una opportuna struttura `data` i dati ad esso relativi, per poi provvedere (24) a stampare il nome del file e la dimensione riportata in `data`.

Dato che la funzione verrà chiamata all'interno di `DirScan` 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.

---

```

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 DirScan:
9  *
10 * Input: the directory name and a computation function
11 * Return: 0 if OK, -1 on errors
12 */
13 int DirScan(char * dirname, int (*compute)(struct dirent *))
14 {
15     DIR * dir;
16     struct dirent *direntry;
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 ((direntry = readdir(dir)) != NULL) { /* read entry */
27         if (compute(direntry)) {           /* execute function on it */
28             return -1;                   /* on error return */
29         }
30     }
31     closedir(dir);
32     return 0;
33 }
```

---

**Figura 5.4:** Codice della routine di scansione di una directory contenuta nel file `DirScan.c`.

Tutto il grosso del lavoro è svolto dalla funzione `DirScan`, riportata in fig. 5.4. 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 (19-23) uno stream sulla directory passata come primo argomento, stampando un messaggio in caso di errore.

Il passo successivo (24-25) è cambiare directory di lavoro (vedi sez. 5.1.7), usando in sequenza le funzione `dirfd` e `fchdir` (in realtà si sarebbe potuto usare direttamente `chdir` su `dirname`),

in modo che durante il successivo ciclo (27-31) sulle singole voci dello stream ci si trovi all'interno della directory.<sup>19</sup>

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 (27) 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 `compare` (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 (32) dello stream<sup>20</sup> e la restituzione (33) del codice di operazioni concluse con successo.

### 5.1.7 La directory di lavoro

A ciascun processo è associata una directory nel filesystem che è chiamata *directory corrente* o *directory di lavoro* (in inglese *current working directory*) che è quella a cui si fa riferimento 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 corrente resta la stessa quando viene creato un processo figlio (vedi sez. 3.2.2), la directory corrente della shell diventa anche la directory corrente di qualunque comando da essa lanciato.

In genere il kernel tiene traccia per ciascun processo dell'inode della directory di lavoro, per ottenere il *pathname* occorre usare una apposita funzione di libreria, `getcwd`, il cui prototipo è:

```
#include <unistd.h>
char *getcwd(char *buffer, size_t size)
    Legge il pathname della directory di lavoro corrente.
```

La funzione restituisce il puntatore `buffer` se riesce, `NULL` se fallisce, in quest'ultimo caso la variabile `errno` è impostata con i seguenti codici di errore:

`EINVAL` L'argomento `size` è zero e `buffer` non è nullo.

`ERANGE` L'argomento `size` è più piccolo della lunghezza del *pathname*.

`EACCES` Manca il permesso di lettura o di ricerca su uno dei componenti del *pathname* (cioè su una delle directory superiori alla corrente).

La funzione restituisce il *pathname* completo della directory di lavoro nella stringa puntata da `buffer`, che deve essere precedentemente allocata, per una dimensione massima di `size`. Il buffer deve essere sufficientemente lungo 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.

Si può anche specificare un puntatore nullo come `buffer`,<sup>21</sup> 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 una volta cessato il suo utilizzo.

<sup>19</sup>questo 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).

<sup>20</sup>nel 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 gli stream comporterebbe un consumo progressivo di risorse, con conseguente rischio di esaurimento delle stesse

<sup>21</sup>questa è un'estensione allo standard POSIX.1, supportata da Linux.

Di questa funzione esiste una versione `char *getwd(char *buffer)` fatta per compatibilità all'indietro con BSD, che non consente di specificare la dimensione del buffer; esso deve essere allocato in precedenza ed avere una dimensione superiore a `PATH_MAX` (di solito 256 byte, vedi sez. 8.1.1); il problema è che in Linux non esiste una dimensione superiore per 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.

Una seconda funzione simile è `char *get_current_dir_name(void)` che è sostanzialmente equivalente ad una `getcwd(NULL, 0)`, con la sola differenza che essa ritorna il valore della variabile di ambiente `PWD`, che essendo costruita dalla shell può contenere un *pathname* comprendente anche dei link simbolici. Usando `getcwd` infatti, essendo il *pathname* ricavato risalendo all'indietro l'albero della directory, si perderebbe traccia di ogni passaggio attraverso eventuali link simbolici.

Per cambiare la directory di lavoro si può usare la funzione `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 in pathname.
```

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

<code>ENOTDIR</code>	Non si è specificata una directory.
<code>EACCES</code>	Manca il permesso di ricerca su uno dei componenti di <code>path</code> .
ed inoltre <code>EFAULT</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> , <code>ENOMEM</code> , <code>ELOOP</code> e <code>EIO</code> .	

ed ovviamente `pathname` deve indicare una directory per la quale si hanno i permessi di accesso.

Dato che anche le directory sono file, è possibile riferirsi ad esse anche tramite il file descriptor, e non solo tramite il *pathname*, per fare questo si usa `fchdir`, il cui prototipo è:

```
#include <unistd.h>
int fchdir(int fd)
    Identica a chdir, ma usa il file descriptor fd invece del pathname.
```

La funzione restituisce zero in caso di successo e -1 per un errore, in caso di errore `errno` assumerà i valori `EBADF` o `EACCES`.

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 accesso alla directory specificata da `fd`.

### 5.1.8 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. 3.5.2).

Le *glibc* provvedono varie funzioni per generare nomi di file temporanei, di cui si abbia certezza di unicità (al momento della generazione); la prima di queste funzioni è `tmpnam` il cui prototipo è:

```
#include <stdio.h>
char *tmpnam(char *string)
    Restituisce il puntatore ad una stringa contenente un nome di file valido e non esistente al momento dell'invocazione.
```

La funzione ritorna il puntatore alla stringa con il nome o `NULL` in caso di fallimento. Non sono definiti errori.

se si è passato un puntatore **string** non nullo questo deve essere di dimensione **L\_tmpnam** (costante definita in **stdio.h**, come **P\_tmpdir** e **TMP\_MAX**) 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. Al nome viene automaticamente aggiunto come prefisso la directory specificata da **P\_tmpdir**.

Di questa funzione esiste una versione rientrante, **tmpnam\_r**, che non fa nulla quando si passa **NULL** come parametro. 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)
    Restituisce il puntatore ad una stringa contenente un nome di file valido e non esistente al momento dell'invocazione.
```

La funzione ritorna il puntatore alla stringa con il nome o **NULL** in caso di fallimento, **errno** viene impostata a **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 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 accessibili), la prima valida delle seguenti:

- La variabile di ambiente **TMPNAME** (non ha effetto se non è definita o se il programma chiamante è **suid** o **sgid**, vedi sez. 5.3.2).
- il valore dell'argomento **dir** (se diverso da **NULL**).
- Il valore della costante **P\_tmpdir**.
- la directory **/tmp**.

In ogni caso, anche se 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 con lo stesso nome; per questo motivo quando si usa il nome ottenuto da una di queste funzioni occorre sempre 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.

Per evitare di dovere effettuare a mano tutti questi controlli, lo standard POSIX definisce la funzione **tmpfile**, il cui prototipo è:

```
#include <stdio.h>
FILE *tmpfile (void)
    Restituisce un file temporaneo aperto in lettura/scrittura.
```

La funzione ritorna il puntatore allo stream associato al file temporaneo in caso di successo e **NULL** in caso di errore, nel qual caso **errno** assumerà i valori:

**EINTR** La funzione è stata interrotta da un segnale.  
**EEXIST** Non è stato possibile generare un nome univoco.  
ed inoltre **EFAULT**, **EMFILE**, **ENFILE**, **ENOSPC**, **EROFS** e **EACCES**.

essa restituisce direttamente uno stream già aperto (in modalità **r+b**, si veda sez. 7.2.1) 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 le *glibc* prima tentano 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 un nome casuale, il suo prototipo è:

```
#include <stlib.h>
char *mktemp(char *template)
    Genera un filename univoco sostituendo le XXXXXX finali di template.
```

La funzione ritorna il puntatore `template` in caso di successo e `NULL` in caso di errore, nel qual caso `errno` assumerà i valori:

`EINVAL` `template` non termina con XXXXXX.

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 uno stream; il suo prototipo è:

```
#include <stlib.h>
int mkstemp(char *template)
    Genera un file temporaneo con un nome ottenuto sostituendo le XXXXXX finali di template.
```

La funzione ritorna il file descriptor in caso successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

`EINVAL` `template` non termina con XXXXXX.

`EEXIST` non è riuscita a creare un file temporaneo, il contenuto di `template` è indefinito.

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. 6.2.1), in questo modo al ritorno della funzione si ha la certezza di essere i soli utenti del file. I permessi sono impostati al valore 0600<sup>22</sup> (si veda sez. 5.3.1).

In OpenBSD è stata introdotta un'altra funzione<sup>23</sup> simile alle precedenti, `mkdtemp`, che crea una directory temporanea; il suo prototipo è:

```
#include <stlib.h>
char *mkdtemp(char *template)
    Genera una directory temporanea il cui nome è ottenuto sostituendo le XXXXXX finali di template.
```

La funzione ritorna il puntatore al nome della directory in caso successo e `NULL` in caso di errore, nel qual caso `errno` assumerà i valori:

`EINVAL` `template` non termina con XXXXXX.

più gli altri eventuali codici di errore di `mkdir`.

la directory è creata con permessi 0700 (al solito si veda cap. 6 per i dettagli); dato che la creazione della directory è sempre esclusiva i precedenti problemi di *race condition* non si pongono.

## 5.2 La manipolazione delle caratteristiche dei files

Come spiegato in sez. 4.2.3 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

<sup>22</sup>questo è vero a partire dalle *glibc* 2.0.7, le versioni precedenti delle *glibc* e le vecchie *libc5* e *libc4* usavano il valore 0666 che permetteva a chiunque di leggere i contenuti del file.

<sup>23</sup>introdotta anche in Linux a partire dalle *glibc* 2.1.91.

varie funzioni usate per manipolare tutte queste informazioni (eccetto quelle che riguardano la gestione del controllo di accesso, trattate in sez. 5.3).

### 5.2.1 Le funzioni stat, fstat e lstat

La lettura delle informazioni relative ai file è fatta attraverso la famiglia delle funzioni **stat** (**stat**, **fstat** e **lstat**); questa è la funzione che ad esempio usa il comando **ls** per poter ottenere e mostrare tutti i dati dei files. I prototipi di queste funzioni sono i seguenti:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf)
    Legge le informazioni del file specificato da file_name e le inserisce in buf.
int lstat(const char *file_name, struct stat *buf)
    Identica a stat eccetto che se il file_name è un link simbolico vengono lette le informazioni relativae ad esso e non al file a cui fa riferimento.
int fstat(int filedes, struct stat *buf)
    Identica a stat eccetto che si usa con un file aperto, specificato tramite il suo file descriptor filedes.
```

Le funzioni restituiscono 0 in caso di successo e -1 per un errore, nel qual caso **errno** assumerà uno dei valori: **EBADF**, **ENOENT**, **ENOTDIR**, **ELOOP**, **EFAULT**, **EACCES**, **ENOMEM**, **ENAMETOOLONG**.

il loro comportamento è identico, solo che operano rispettivamente su un file, su un link simbolico e su un file descriptor.

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. 5.5, 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 più precisi, o per il padding dei campi).

---

```
struct stat {
    dev_t          st_dev;        /* device */
    ino_t          st_ino;        /* inode */
    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 */
    unsigned long  st_blksize;   /* blocksize for filesystem I/O */
    unsigned long  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 change */
};
```

---

**Figura 5.5:** 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**).

### 5.2.2 I tipi di file

Come riportato in tab. 4.1 in Linux oltre ai file e alle directory esistono altri oggetti che possono stare su un filesystem. Il tipo di file è ritornato dalla **stat** come maschera binaria nel campo

`st_mode` (che contiene anche le informazioni relative ai permessi).

Dato che il valore numerico può variare a seconda delle implementazioni, lo standard POSIX definisce un insieme di macro per verificare il tipo di file, queste vengono usate anche da Linux che supporta pure le estensioni allo standard per i link simbolici e i socket definite da BSD; l'elenco completo delle macro con cui è possibile estrarre l'informazione da `st_mode` è riportato in tab. 5.3.

Macro	Tipo del file
<code>S_ISREG(m)</code>	file regolare
<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>	fifo
<code>S_ISLNK(m)</code>	link simbolico
<code>S_ISSOCK(m)</code>	socket

**Tabella 5.3:** Macro per i tipi di file (definite in `sys/stat.h`).

Oltre alle macro di tab. 5.3 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 costanti numeriche riportate in tab. 5.4.

Il primo valore dell'elenco di tab. 5.4 è la maschera binaria che permette di estrarre i bit nei quali viene memorizzato il tipo di file, i valori successivi sono le costanti corrispondenti ai singoli bit, e possono essere usati per effettuare la selezione sul tipo di file voluto, con un'opportuna combinazione.

Flag	Valore	Significato
<code>S_IFMT</code>	0170000	maschera per i bit del tipo di file
<code>S_IFSOCK</code>	0140000	socket
<code>S_IFLNK</code>	0120000	link simbolico
<code>S_IFREG</code>	0100000	file regolare
<code>S_IFBLK</code>	0060000	dispositivo a blocchi
<code>S_IFDIR</code>	0040000	directory
<code>S_IFCHR</code>	0020000	dispositivo a caratteri
<code>S_IFIFO</code>	0010000	fifo
<code>S_ISUID</code>	0004000	set UID bit
<code>S_ISGID</code>	0002000	set GID bit
<code>S_ISVTX</code>	0001000	sticky bit
<code>S_IRUSR</code>	00400	il proprietario ha permesso di lettura
<code>S_IWUSR</code>	00200	il proprietario ha permesso di scrittura
<code>S_IXUSR</code>	00100	il proprietario ha permesso di esecuzione
<code>S_IRGRP</code>	00040	il gruppo ha permesso di lettura
<code>S_IWGRP</code>	00020	il gruppo ha permesso di scrittura
<code>S_IXGRP</code>	00010	il gruppo ha permesso di esecuzione
<code>S_IROTH</code>	00004	gli altri hanno permesso di lettura
<code>S_IWOTH</code>	00002	gli altri hanno permesso di esecuzione
<code>S_IXOTH</code>	00001	gli altri hanno permesso di esecuzione

**Tabella 5.4:** Costanti per l'identificazione dei vari bit che compongono il campo `st_mode` (definite in `sys/stat.h`).

Ad esempio se si volesse impostare una condizione che permetta di controllare se un file è una directory o un file ordinario si potrebbe definire la macro di preprocessore:

```
#define IS_FILE_DIR(x) (((x) & S_IFMT) & (S_IFDIR | S_IFREG))
```

in cui prima si estraggono da `st_mode` i bit relativi al tipo di file e poi si effettua il confronto con la combinazione di tipi scelta.

### 5.2.3 Le dimensioni dei file

Il campo **st\_size** contiene la dimensione del file in byte (se si tratta di un file regolare, nel caso di un link simbolico la dimensione è quella del *pathname* che contiene, per le fifo è sempre nullo).

Il campo **st\_blocks** definisce la lunghezza del file in blocchi di 512 byte. Il campo **st\_blksize** infine definisce la dimensione preferita per i trasferimenti sui file (che è la dimensione usata anche dalle librerie del C per l'interfaccia degli stream); scrivere sul file a blocchi di dati di dimensione inferiore sarebbe inefficiente.

Si tenga conto che la lunghezza del file riportata in **st\_size** non è detto che corrisponda all'occupazione dello spazio su disco per via della possibile esistenza dei cosiddetti *holes* (letteralmente *buchi*) che si formano tutte le volte che si va a scrivere su un file dopo aver eseguito una **lseek** (vedi sez. 6.2.3) oltre la sua fine.

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 dal file (ad esempio usando il comando **wc -c**), dato che in tal caso per le parti non scritte vengono restituiti degli zeri, si avrà lo stesso risultato di **ls**.

Se è sempre possibile allargare un file, scrivendoci sopra od usando la funzione **lseek** 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**, ma questo è un caso particolare; per qualunque altra dimensione si possono usare le due funzioni **truncate** e **ftruncate**, i cui prototipi sono:

```
#include <unistd.h>
int truncate(const char *file_name, off_t length)
    Fa sì che la dimensione del file file_name sia troncata ad un valore massimo specificato da lenght.
int ftruncate(int fd, off_t length)
    Identica a truncate eccetto che si usa con un file aperto, specificato tramite il suo file descriptor fd.
```

Le funzioni restituiscono zero in caso di successo e -1 per un errore, nel qual caso **errno** viene impostata opportunamente; per **ftruncate** si hanno i valori:

<b>EBADF</b>	<b>fd</b> non è un file descriptor.
<b>EINVAL</b>	<b>fd</b> è un riferimento ad un socket, non a un file o non è aperto in scrittura.
per <b>truncate</b> si hanno:	
<b>EACCES</b>	il file non ha permesso di scrittura o non si ha il permesso di esecuzione una delle directory del <i>pathname</i> .
<b>ETXTBSY</b>	Il file è un programma in esecuzione.
ed anche <b>ENOTDIR</b> , <b>ENAMETOOLONG</b> , <b>ENOENT</b> , <b>EROFS</b> , <b>EIO</b> , <b>EFAULT</b> , <b>ELOOP</b> .	

Se il file è più lungo della lunghezza specificata i dati in eccesso saranno perduti; il comportamento in caso di lunghezza inferiore non è specificato e dipende dall'implementazione: il file può essere lasciato invariato o esteso fino alla lunghezza scelta; in quest'ultimo caso lo spazio viene riempito con zeri (e in genere si ha la creazione di un *hole* nel file).

### 5.2.4 I tempi dei file

Il sistema mantiene per ciascun file tre tempi. Questi sono registrati nell'inode insieme agli altri attributi del file e possono essere letti tramite la funzione **stat**, che li restituisce attraverso tre campi della struttura **stat** di fig. 5.5. Il significato di detti tempi e dei relativi campi è riportato nello schema in tab. 5.5, dove è anche riportato un esempio delle funzioni che effettuano cambiamenti su di essi.

Membro	Significato	Funzione	Opzione di ls
<code>st_atime</code>	ultimo accesso ai dati del file	<code>read, utime</code>	<code>-u</code>
<code>st_mtime</code>	ultima modifica ai dati del file	<code>write, utime</code>	default
<code>st_ctime</code>	ultima modifica ai dati dell'inode	<code>chmod, utime</code>	<code>-c</code>

**Tabella 5.5:** I tre tempi associati a ciascun file.

Il primo punto da tenere presente è la differenza fra il cosiddetto tempo di modifica (il *modification time* `st_mtime`) e il tempo di cambiamento di stato (il *change time* `st_ctime`). Il primo infatti fa riferimento ad una modifica del contenuto di un file, mentre il secondo ad una modifica dell'inode; siccome esistono molte operazioni (come la funzione `link` e molte altre che vedremo in seguito) che modificano solo le informazioni contenute nell'inode senza toccare il file, diventa necessario l'utilizzo di un altro tempo.

Il sistema non tiene conto dell'ultimo accesso all'inode, pertanto funzioni come `access` o `stat` non hanno alcuna influenza sui tre tempi. Il tempo di ultimo accesso (ai dati) viene di solito usato per cancellare i file che non servono più dopo un certo lasso di tempo (ad esempio `leafnode` cancella i vecchi articoli sulla base di questo tempo).

Il tempo di ultima modifica invece viene usato da `make` per decidere quali file necessitano di essere ricompilati o (talvolta insieme anche al tempo di cambiamento di stato) per decidere quali file devono essere archiviati per il backup. 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. 5.5.

Funzione	File o directory del riferimento			Directory contenente il riferimento			Note
	(a)	(m)	(c)	(a)	(m)	(c)	
<code>chmod, fchmod</code>	—	—	●	—	—	—	
<code>chown, fchown</code>	—	—	●	—	—	—	
<code>creat</code>	●	●	●	—	●	●	
<code>creat</code>	—	●	●	—	●	●	
<code>exec</code>	●	—	—	—	—	—	
<code>lchown</code>	—	—	●	—	—	—	
<code>link</code>	—	—	●	—	●	●	
<code>mkdir</code>	●	●	●	—	●	●	
<code>mkfifo</code>	●	●	●	—	●	●	
<code>open</code>	●	●	●	—	●	●	con <code>O_CREATE</code>
<code>open</code>	—	●	●	—	—	—	con <code>O_TRUNC</code>
<code>pipe</code>	●	●	●	—	—	—	
<code>read</code>	●	—	—	—	—	—	
<code>remove</code>	—	—	●	—	●	●	se esegue <code>unlink</code>
<code>remove</code>	—	—	—	—	●	●	se esegue <code>rmdir</code>
<code>rename</code>	—	—	●	—	●	●	per entrambi gli argomenti
<code>rmdir</code>	—	—	—	—	●	●	
<code>truncate, ftruncate</code>	—	●	●	—	—	—	
<code>unlink</code>	—	—	●	—	●	●	
<code>utime</code>	●	●	●	—	—	—	
<code>write</code>	—	●	●	—	—	—	

**Tabella 5.6:** Prospetto dei cambiamenti effettuati sui tempi di ultimo accesso (a), ultima modifica (m) e ultimo cambiamento (c) dalle varie funzioni operanti su file e directory.

L'effetto delle varie funzioni di manipolazione dei file sui tempi è illustrato in tab. 5.6. 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 se si tiene conto di quanto già detto, e cioè che anche le directory sono 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 modifica. Un esempio di questo 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 noti infine come `st_ctime` non abbia nulla a che fare con il tempo di creazione del file, usato in molti altri sistemi operativi, ma che in Unix non esiste. Per questo motivo 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 come data di ultima modifica.

### 5.2.5 La funzione `utime`

I tempi di ultimo accesso e modifica possono essere cambiati usando la funzione `utime`, il cui prototipo è:

```
#include <utime.h>
int utime(const char *filename, struct utimbuf *times)
    Cambia i tempi di ultimo accesso e modifica dell'inode specificato da filename secondo i
    campi actime e modtime di times. Se questa è NULL allora viene usato il tempo corrente.

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso errno assumerà
uno dei valori:
EACCES    non si ha il permesso di scrittura sul file.
ENOENT    filename non esiste.
```

La funzione prende come argomento `times` una struttura `utimbuf`, la cui definizione è riportata in fig. 5.6, con la quale si possono specificare i nuovi valori che si vogliono impostare per tempi.

```
struct utimbuf {
    time_t actime; /* access time */
    time_t modtime; /* modification time */
};
```

*Figura 5.6:* La struttura `utimbuf`, usata da `utime` per modificare i tempi dei file.

L'effetto della funzione e i privilegi necessari per eseguirla dipendono da cosa è l'argomento `times`; se è `NULL` la funzione imposta il tempo corrente ed è sufficiente avere accesso in scrittura al file; se invece si è specificato un valore la funzione avrà successo solo se si è proprietari del file (o si hanno i privilegi di amministratore).

Si tenga presente che non è comunque possibile specificare il tempo di cambiamento di stato del file, che viene comunque cambiato dal kernel tutte le volte che si modifica l'inode (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 molto più complicata da realizzare.

## 5.3 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.<sup>24</sup> In questa sezione ne

<sup>24</sup>per standard si intende che implementa le caratteristiche previste dallo standard POSIX. In Linux sono disponibili anche una serie di altri filesystem, come quelli di Windows e del Mac, che non supportano queste

esamineremo i concetti essenziali e le funzioni usate per gestirne i vari aspetti.

### 5.3.1 I permessi per l'accesso ai file

Ad ogni file Linux associa sempre l'utente che ne è proprietario (il cosiddetto *owner*) ed un gruppo di appartenenza, secondo il meccanismo degli identificatori di utente e gruppo (*uid* e *gid*). Questi valori sono accessibili da programma tramite la funzione `stat`, e sono mantenuti nei campi `st_uid` e `st_gid` della struttura `stat` (si veda sez. 5.2.1).<sup>25</sup>

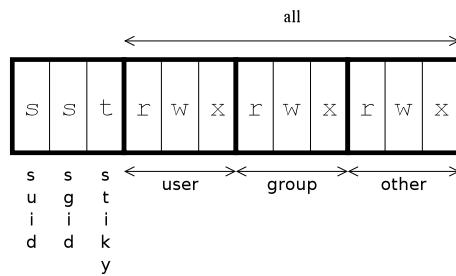
Il controllo di accesso ai file segue un modello abbastanza semplice che prevede tre permessi fondamentali strutturati su tre livelli di accesso. Esistono varie estensioni a questo modello,<sup>26</sup> ma nella maggior parte dei casi il meccanismo standard è più che sufficiente a soddisfare tutte le 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 rispettivamente al proprietario, al gruppo, a tutti gli altri.



**Figura 5.7:** Lo schema dei bit utilizzati per specificare i permessi di un file contenuti nel campo `st_mode` di `fstat`.

I restanti tre bit (noti come *suid*, *sgid*, e *sticky*) sono usati per indicare alcune caratteristiche più complesse del meccanismo del controllo di accesso su cui torneremo in seguito (in sez. 5.3.2 e sez. 5.3.3); lo schema di allocazione dei bit è riportato in fig. 5.7.

Anche i permessi, come tutte le altre informazioni pertinenti al file, sono memorizzati nell'i-node; in particolare essi sono contenuti in alcuni bit del campo `st_mode` della struttura `stat` (si veda di nuovo fig. 5.5).

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

---

caratteristiche.

<sup>25</sup>Questo 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 i permessi vengono assegnati in maniera fissa con un opzione in fase di montaggio.

<sup>26</sup>Come le *Access Control List* che possono essere aggiunte al filesystem standard con opportune patch, la cui introduzione nei kernel ufficiali è iniziata con la serie 2.5.x. per arrivare a meccanismi di controllo ancora più sofisticati come il *mandatory access control* di SE-Linux.

uso 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` sono riportate in tab. 5.7.

<code>st_mode</code> bit	Significato
<code>S_IRUSR</code>	<i>user-read</i> , l'utente può leggere
<code>S_IWUSR</code>	<i>user-write</i> , l'utente può scrivere
<code>S_IXUSR</code>	<i>user-execute</i> , l'utente può eseguire
<code>S_IRGRP</code>	<i>group-read</i> , il gruppo può leggere
<code>S_IWGRP</code>	<i>group-write</i> , il gruppo può scrivere
<code>S_IXGRP</code>	<i>group-execute</i> , il gruppo può eseguire
<code>S_IROTH</code>	<i>other-read</i> , tutti possono leggere
<code>S_IWOTH</code>	<i>other-write</i> , tutti possono scrivere
<code>S_IXOTH</code>	<i>other-execute</i> , tutti possono eseguire

**Tabella 5.7:** I bit dei permessi di accesso ai file, come definiti in `<sys/stat.h>`

I permessi vengono usati in maniera diversa dalle varie funzioni, e a seconda che si riferiscano a dei file, dei link 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*, ed è 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 in una directory (se si hanno i permessi opportuni per il medesimo) ma non si potrà vederlo con `ls` (mentre per crearlo occorrerà 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 tab. 6.2) 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.

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), non è necessario nessun tipo di permesso per il file stesso (infatti esso non viene toccato, viene solo modificato il contenuto della directory, rimuovendo la voce che ad esso fa riferimento).

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 avere il permesso di esecuzione, inoltre solo i file regolari possono essere eseguiti.

I permessi per un link simbolico sono ignorati, contano quelli del file a cui fa riferimento; per questo in genere il comando `ls` riporta per un link simbolico tutti i permessi come concessi; utente e gruppo a cui esso appartiene vengono pure ignorati quando il link viene risolto, vengono controllati solo quando viene richiesta la rimozione del link e quest'ultimo è in una directory con lo *sticky bit* impostato (si veda sez. 5.3.3).

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'user-ID effettivo, il group-ID effettivo e gli eventuali group-ID supplementari del processo.<sup>27</sup>

<sup>27</sup>in realtà Linux, per quanto riguarda l'accesso ai file, utilizza gli identificatori del gruppo *filesystem* (si ricordi

Per una spiegazione dettagliata degli identificatori associati ai processi si veda sez. 3.3; normalmente, a parte quanto vedremo in sez. 5.3.2, l'user-ID effettivo e il group-ID effettivo corrispondono ai valori dell'*uid* e del *gid* dell'utente che ha lanciato il processo, mentre i group-ID 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:

1. Se l'user-ID effettivo del processo è zero (corrispondente all'amministratore) l'accesso è sempre garantito senza nessun ulteriore controllo. Per questo motivo *root* ha piena libertà di accesso a tutti i file.
2. Se l'user-ID 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 relativo<sup>28</sup> bit dei permessi d'accesso dell'utente è impostato, l'accesso è consentito
  - altrimenti l'accesso è negato
3. Se il group-ID effettivo del processo o uno dei group-ID supplementari dei processi 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.

### 5.3.2 I bit *suid* e *sgid*

Come si è accennato (in sez. 5.3.1) 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. 3.2.7, 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<sup>29</sup>) ha il bit *suid* impostato, il kernel assegnerà come user-ID 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 group-ID effettivo del processo.

---

quanto esposto in sez. 3.3), ma essendo questi del tutto equivalenti ai primi, eccetto il caso in cui si voglia scrivere un server NFS, ignoreremo questa differenza.

<sup>28</sup>per relativo si intende il bit di user-read se il processo vuole accedere in scrittura, quello di user-write per l'accesso in scrittura, etc.

<sup>29</sup>per motivi di sicurezza il kernel ignora i bit *suid* e *sgid* per gli script eseguibili.

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 a root ma ha il bit *suid* impostato per cui quando viene lanciato da un utente normale parte con i privilegi di root.

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. 3.3).

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_IGID`, i cui valori sono riportati in tab. 5.4.

Gli stessi bit vengono ad assumere in 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. 5.3.4 per una spiegazione dettagliata al proposito).

Infine Linux utilizza il bit *sgid* per una 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. 11.4.5).

### 5.3.3 Il bit *sticky*

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 files 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 continuo indicizzato direttamente in questo modo si poteva risparmiare in tempo di caricamento rispetto alla ricerca del file su disco. 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;<sup>30</sup> 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

---

<sup>30</sup>lo *sticky bit* per le directory è un'estensione non definita nello standard POSIX, Linux però la supporta, così come BSD e SVr4.

un classico esempio di directory che ha questo bit impostato è `/tmp`, i permessi infatti di solito sono i seguenti:

```
$ ls -ld /tmp
drwxrwxrwt  6 root      root        1024 Aug 10 01:03 /tmp
```

quindi con lo *sticky 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.

### 5.3.4 La titolarità di nuovi file e directory

Vedremo in sez. 6.2 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. 5.1.4).

Lo standard POSIX prescrive che l'*uid* del nuovo file corrisponda all'user-ID effettivo del processo che lo crea; per il *gid* invece prevede due diverse possibilità:

- il *gid* del file corrisponde al group-ID effettivo del processo.
- il *gid* del file corrisponde 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 quella che viene chiamata semantica SVr4; di norma cioè il nuovo file viene creato, seguendo la prima opzione, con il *gid* del processo, se però la directory in cui viene creato il file ha il bit *sgid* impostato allora viene usata la seconda opzione.

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 BSD necessita che per le nuove directory venga anche propagato anche il bit *sgid*. Questo è il comportamento predefinito del comando `mkdir`, ed è in questo modo ad esempio che Debian assicura che le sotto-directory create nella home di un utente restino sempre con il *gid* del gruppo primario dello stesso.

### 5.3.5 La funzione access

Come visto in sez. 5.3 il controllo di accesso ad un file viene fatto utilizzando l'user-ID ed il group-ID effettivo del processo; ci sono casi però in cui si può voler effettuare il controllo con l'user-ID reale ed il group-ID 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. 5.3.2 e spiegato in dettaglio in sez. 3.3, non è detto siano uguali a quelli effettivi.

Per far questo si può usare la funzione `access`, il cui prototipo è:

```
#include <unistd.h>
int access(const char *pathname, int mode)
    Verifica i permessi di accesso.
```

La funzione ritorna 0 se l'accesso è consentito, -1 se l'accesso non è consentito ed in caso di errore; nel qual caso la variabile `errno` assumerà i valori:

<code>EINVAL</code>	il valore di <code>mode</code> non è valido.
<code>EACCES</code>	l'accesso al file non è consentito, o non si ha il permesso di attraversare una delle directory di <code>pathname</code> .
<code>EROFS</code>	si è richiesto l'accesso in scrittura per un file su un filesystem montato in sola lettura. ed inoltre <code>EFAULT</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> , <code>ENOTDIR</code> , <code>ELOOP</code> , <code>EIO</code> .

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. 5.8 (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 link 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, si ricordi che il fatto che una directory abbia permesso di scrittura non significa che ci si possa scrivere come in un file, e il fatto che un file abbia permesso di esecuzione non comporta che contenga un programma eseguibile. La funzione ritorna zero solo se tutti i permessi controllati sono disponibili, in caso contrario (o di errore) ritorna -1.

<code>mode</code>	Significato
<code>R_OK</code>	verifica il permesso di lettura
<code>W_OK</code>	verifica il permesso di scrittura
<code>X_OK</code>	verifica il permesso di esecuzione
<code>F_OK</code>	verifica l'esistenza del file

**Tabella 5.8:** 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.

### 5.3.6 Le funzioni `chmod` e `fchmod`

Per cambiare i permessi di un file il sistema mette ad disposizione due funzioni `chmod` e `fchmod`, che operano rispettivamente su un filename e su un 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.
```

Le funzioni restituiscono zero in caso di successo e -1 per un errore, in caso di errore `errno` può assumere i valori:

<code>EPERM</code>	L'user-ID effettivo non corrisponde a quello del proprietario del file o non è zero.
<code>EROFS</code>	Il file è su un filesystem in sola lettura.
ed inoltre <code>EIO</code> ; <code>chmod</code> restituisce anche <code>EFAULT</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> , <code>ENOMEM</code> , <code>ENOTDIR</code> , <code>EACCES</code> , <code>ELLOOP</code> ; <code>fchmod</code> anche <code>EBADF</code> .	

Entrambe le funzioni utilizzano come secondo argomento `mode`, una variabile dell'apposito tipo primitivo `mode_t` (vedi tab. 1.2) utilizzato per specificare i permessi sui file.

Le costanti con cui specificare i singoli bit di `mode` sono riportate in tab. 5.9. 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, con un valore numerico (la shell lo vuole in ottale, dato che i bit dei permessi sono divisibili in gruppi di tre), che si può calcolare direttamente usando lo schema si utilizzo dei bit illustrato in fig. 5.7.

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 di 0755, se si volesse attivare il bit *suid* il valore da fornire sarebbe 4755.

mode	Valore	Significato
S_ISUID	04000	set user ID
S_ISGID	02000	set group ID
S_ISVTX	01000	sticky bit
S_IRWXU	00700	l'utente ha tutti i permessi
S_IRUSR	00400	l'utente ha il permesso di lettura
S_IWUSR	00200	l'utente ha il permesso di scrittura
S_IXUSR	00100	l'utente ha il permesso di esecuzione
S_IRWXG	00070	il gruppo ha tutti i permessi
S_IRGRP	00040	il gruppo ha il permesso di lettura
S_IWGRP	00020	il gruppo ha il permesso di scrittura
S_IXGRP	00010	il gruppo ha il permesso di esecuzione
S_IRWXO	00007	gli altri hanno tutti i permessi
S_IROTH	00004	gli altri hanno il permesso di lettura
S_IWOTH	00002	gli altri hanno il permesso di scrittura
S_IXOTH	00001	gli altri hanno il permesso di esecuzione

**Tabella 5.9:** Valori delle costanti usate per indicare i vari bit di mode utilizzato per impostare i permessi dei file.

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'user-ID effettivo del processo corrisponde a quello del proprietario del file o dell'amministratore, altrimenti 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:

1. siccome solo l'amministratore può impostare lo *sticky bit*, se l'user-ID 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. 5.3.4 riguardo la creazione dei nuovi file, si può avere il caso in cui il file creato da un processo è assegnato a un gruppo per il quale il processo non ha privilegi. Per evitare che si possa assegnare il bit *sgid* ad un file appartenente a 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'user-ID effettivo del processo è zero).

Per alcuni filesystem<sup>31</sup> è inoltre prevista una 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'amministratore 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.

### 5.3.7 La funzione umask

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. 6.2.1, 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.

In tutti questi casi 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

<sup>31</sup>il filesystem *ext2* supporta questa caratteristica, che è mutuata da BSD.

tre i permessi di sez. 5.3.1 (cioè 666 nel primo caso e 222 nel secondo). Per questo motivo il sistema associa ad ogni processo<sup>32</sup> 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.

La funzione 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 dei bit al valore specificato da mask (di cui vengono presi solo i 9 bit meno significativi).
```

La funzione ritorna il precedente valore della maschera. È una delle poche funzioni che non restituisce codici di errore.

In genere si usa questa maschera per impostare un valore predefinito che escluda preventivamente alcuni permessi (usualmente quello 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, e gli utenti non hanno motivi per modificarlo.

### 5.3.8 Le funzioni **chown**, **fchown** e **lchown**

Come per i permessi, il sistema fornisce anche delle funzioni che permettano di cambiare utente e gruppo cui il file appartiene; le funzioni in questione sono tre: **chown**, **fchown** e **lchown**, ed i loro 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)
    Le funzioni cambiano utente e gruppo di appartenenza di un file ai valori specificati dalle variabili owner e group.
```

Le funzioni restituiscono zero in caso di successo e -1 per un errore, in caso di errore **errno** può assumere i valori:

**EPERM** L'utente-ID effettivo non corrisponde a quello del proprietario del file o non è zero, o utente e gruppo non sono validi

Oltre a questi entrambe restituiscono gli errori **EROFS** e **EIO**; **chown** restituisce anche **EFAULT**, **ENAMETOOLONG**, **ENOENT**, **ENOMEM**, **ENOTDIR**, **EACCES**, **ELOOP**; **fchown** anche **EBADF**.

In Linux soltanto l'amministratore può cambiare il proprietario di un file, seguendo la semantica di BSD che non consente agli utenti di assegnare i loro file ad altri (per evitare eventuali aggiramenti delle quote). L'amministratore può cambiare il gruppo di un file, il proprietario può cambiare il gruppo dei file che gli appartengono solo se il nuovo gruppo è il suo gruppo primario o uno dei gruppi a cui appartiene.

La funzione **chown** segue i link simbolici, per operare direttamente su un link simbolico si deve usare la funzione **lchown**.<sup>33</sup> 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.

<sup>32</sup>è infatti contenuta nel campo **umask** della struttura **fs\_struct**, vedi fig. 3.2.

<sup>33</sup>fino alla versione 2.1.81 in Linux **chown** non seguiva i link 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 link simbolici.

Quando queste funzioni sono chiamate con successo da un processo senza i privilegi di root 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*.

### 5.3.9 Un quadro d'insieme sui permessi

Avendo affrontato in maniera separata il comportamento delle varie funzioni ed il significato dei singoli bit dei permessi sui file, vale la pena fare un riepilogo in cui si riassumono le caratteristiche di ciascuno di essi, in modo da poter fornire un quadro d'insieme.

In tab. 5.10 si sono riassunti gli effetti dei vari bit per un file; per quanto riguarda l'applicazione dei permessi per proprietario, gruppo ed altri si ricordi quanto illustrato in sez. 5.3.1. Si rammenti che il valore dei permessi non ha alcun effetto qualora il processo possieda i privilegi di amministratore.

			user			group			other			Operazioni possibili
s	s	t	r	w	x	r	w	x	r	w	x	
1	-	-	-	-	-	-	-	-	-	-	-	Se eseguito ha i permessi del proprietario
-	1	-	-	-	1	-	-	-	-	-	-	Se eseguito ha i permessi del gruppo proprietario
-	1	-	-	-	0	-	-	-	-	-	-	Il <i>mandatory locking</i> è abilitato
-	-	1	-	-	-	-	-	-	-	-	-	Non utilizzato
-	-	-	1	-	-	-	-	-	-	-	-	Permesso di lettura per il proprietario
-	-	-	-	1	-	-	-	-	-	-	-	Permesso di scrittura per il proprietario
-	-	-	-	-	1	-	-	-	-	-	-	Permesso di esecuzione per il proprietario
-	-	-	-	-	-	1	-	-	-	-	-	Permesso di lettura per il gruppo proprietario
-	-	-	-	-	-	-	1	-	-	-	-	Permesso di scrittura per il gruppo proprietario
-	-	-	-	-	-	-	-	1	-	-	-	Permesso di esecuzione per il gruppo proprietario
-	-	-	-	-	-	-	-	-	1	-	-	Permesso di lettura per tutti gli altri
-	-	-	-	-	-	-	-	-	-	1	-	Permesso di scrittura per tutti gli altri
-	-	-	-	-	-	-	-	-	-	-	1	Permesso di esecuzione per tutti gli altri

**Tabella 5.10:** Tabella riassuntiva del significato dei bit dei permessi per un file.

Per compattezza, nella tabella si sono specificati i bit di *suid*, *sgid* e *sticky* con la notazione illustrata anche in fig. 5.7.

In tab. 5.11 si sono invece riassunti gli effetti dei vari bit dei permessi per una directory; anche in questo caso si sono specificati i bit di *suid*, *sgid* e *sticky* con la notazione compatta illustrata in fig. 5.7.

			user			group			other			Operazioni possibili
s	s	t	r	w	x	r	w	x	r	w	x	
1	-	-	-	-	-	-	-	-	-	-	-	Non utilizzato
-	1	-	-	-	-	-	-	-	-	-	-	Propaga il gruppo proprietario ai nuovi file creati
-	-	1	-	-	-	-	-	-	-	-	-	Limita l'accesso in scrittura dei file nella directory
-	-	-	1	-	-	-	-	-	-	-	-	Permesso di visualizzazione per il proprietario
-	-	-	-	1	-	-	-	-	-	-	-	Permesso di aggiornamento per il proprietario
-	-	-	-	-	1	-	-	-	-	-	-	Permesso di attraversamento per il proprietario
-	-	-	-	-	-	1	-	-	-	-	-	Permesso di visualizzazione per il gruppo proprietario
-	-	-	-	-	-	-	1	-	-	-	-	Permesso di aggiornamento per il gruppo proprietario
-	-	-	-	-	-	-	-	1	-	-	-	Permesso di attraversamento per il gruppo proprietario
-	-	-	-	-	-	-	-	-	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 5.11:** Tabella riassuntiva del significato dei bit dei permessi per una directory.

Nelle tabelle si è indicato con “-” il fatto che il valore degli altri bit non è influente rispetto

a quanto indicato in ciascuna riga; l'operazione fa riferimento soltanto alla combinazione di bit per i quali il valore è riportato esplicitamente.

### 5.3.10 La funzione 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. 3.2.2 ogni processo oltre ad una directory di lavoro, ha anche una directory *radice*<sup>34</sup> che, pur essendo di norma corrispondente alla radice dell'albero di file e directory come visto dal kernel (ed illustrato in sez. 4.1.1), ha per il processo il significato specifico di directory rispetto alla quale vengono risolti i *pathname* assoluti.<sup>35</sup> 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 anche con la radice del filesystem usata dal kernel, e dato che il suo valore viene ereditato dal padre da ogni processo figlio, in generale i processi risolvono i *pathname* assoluti a partire sempre dalla stessa directory, che corrisponde alla / del sistema.

In certe situazioni però, per motivi di sicurezza, è utile poter impedire che un processo possa accedere a tutto il filesystem; per far questo si può cambiare la sua directory radice con la funzione `chroot`, il cui prototipo è:

```
#include <unistd.h>
int chroot(const char *path)
    Cambia la directory radice del processo a quella specificata da path.

La funzione restituisce zero in caso di successo e -1 per un errore, in caso di errore errno può assumere i valori:
EPERM      L'user-ID effettivo del processo non è zero.
ed inoltre EFAULT, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EACCES, ELOOP; EROFS e EIO.
```

in questo modo la directory radice del processo diventerà `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*.

Solo un processo con i privilegi di amministratore può usare questa funzione, e la nuova radice, per quanto detto in sez. 3.2.2, sarà ereditata da tutti i suoi processi figli. Si tenga presente però che la funzione non cambia la directory di lavoro, che potrebbe restare fuori dalla *chroot jail*.

Questo è il motivo per cui la funzione è efficace solo se dopo averla eseguita si cedono i privilegi di root. Infatti se per un qualche motivo il processo resta con la directory di lavoro fuori dalla *chroot jail*, potrà comunque accedere a tutto il resto del filesystem usando *pathname* relativi, i quali, partendo dalla directory di lavoro che è fuori della *chroot jail*, potranno (con l'uso di ...) risalire fino alla radice effettiva del filesystem.

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. Basta infatti creare una nuova *chroot jail* con l'uso di `chroot` su una qualunque directory contenuta nell'attuale directory di lavoro.

<sup>34</sup>entrambe sono contenute in due campi (rispettivamente `pwd` e `root`) di `fs_struct`; vedi fig. 3.2.

<sup>35</sup>cioè quando un processo chiede la risoluzione di un *pathname*, il kernel usa sempre questa directory come punto di partenza.

Per questo motivo l'uso di questa funzione non ha molto senso quando un processo necessita dei privilegi di root per le sue normali operazioni.

Un caso tipico di uso di `chroot` è quello di un server FTP anonimo, in questo caso infatti si vuole che il server veda solo i file che deve trasferire, per cui in genere si esegue una `chroot` sulla directory che contiene i file. Si tenga presente però che in questo caso occorrerà replicare all'interno della *chroot jail* tutti i file (in genere programmi e librerie) di cui il server potrebbe avere bisogno.

# Capitolo 6

## I file: l'interfaccia standard Unix

Esamineremo in questo capitolo la prima delle due interfacce di programmazione per i file, quella dei *file descriptor*, nativa di Unix. Questa è l'interfaccia di basso livello provvista direttamente dalle system call, che non prevede funzionalità evolute come la bufferizzazione o funzioni di lettura o scrittura formattata, e sulla quale è costruita anche l'interfaccia definita dallo standard ANSI C che affronteremo al cap. 7.

### 6.1 L'architettura di base

In questa sezione faremo una breve introduzione sull'architettura su cui è basata dell'interfaccia dei *file descriptor*, che, sia pure con differenze nella realizzazione pratica, resta sostanzialmente la stessa in tutte le implementazioni di un sistema unix-like.

#### 6.1.1 L'architettura 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 (si ricordi quanto visto in sez. 4.2.2). Questo si fa aprendo il file con la funzione `open` che provvederà a localizzare l'inode del file e inizializzare i puntatori che rendono disponibili le funzioni che il VFS mette a disposizione (riportate in 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 intero non negativo, chiamato appunto *file descriptor*. Quando un file viene aperto la funzione `open` restituisce questo numero, tutte le ulteriori operazioni saranno compiute specificando questo stesso valore 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. Il kernel mantiene sempre un elenco dei processi attivi nella cosiddetta *process table* ed un elenco dei file aperti nella *file table*.

La *process table* è una tabella che contiene una voce per ciascun processo attivo nel sistema. In Linux ciascuna voce è costituita da 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`, in cui sono contenute le informazioni relative ai file che il processo ha aperto, ed in particolare:

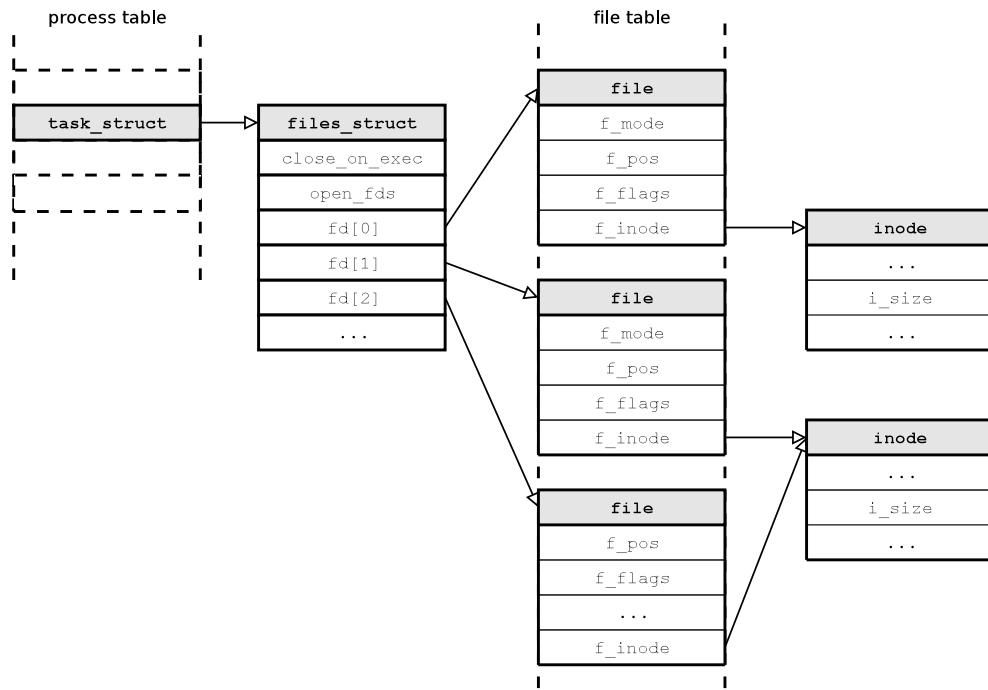
- i flag relativi ai file descriptor.
- il numero di file aperti.
- una tabella che contiene un puntatore alla relativa voce nella *file table* per ogni file aperto.

il *file descriptor* in sostanza è l'intero positivo che indica quest'ultima tabella.

La *file table* è una tabella che contiene una voce per ciascun file che è stato aperto nel sistema. In Linux è costituita da strutture di tipo `file`; in ciascuna di esse sono tenute varie informazioni relative al file, fra cui:

- lo stato del file (nel campo `f_flags`).
- il valore della posizione corrente (*l'offset*) nel file (nel campo `f_pos`).
- un puntatore all'`inode`<sup>1</sup> del file.

In fig. 6.1 si è riportato uno schema in cui è illustrata questa architettura, ed in cui si sono evidenziate le interrelazioni fra le varie strutture di dati sulla quale essa è basata. Ritorneremo su questo schema più volte, dato che esso è fondamentale per capire i dettagli del funzionamento dell'interfaccia dei *file descriptor*.



**Figura 6.1:** Schema della architettura dell'accesso ai file attraverso l'interfaccia dei *file descriptor*.

### 6.1.2 I file standard

Come accennato i *file descriptor* non sono altro che un indice nella tabella dei file aperti di ciascun processo; per questo motivo essi vengono assegnati in successione tutte le volte che si apre un nuovo file (se non ne è stato chiuso nessuno in precedenza).

In tutti i sistemi unix-like esiste una convenzione generale per cui ogni processo viene lanciato dalla shell con almeno tre file aperti. Questi, per quanto appena detto, avranno come *file descriptor* i valori 0, 1 e 2. Benché questa sia soltanto una convenzione, essa è seguita dalla gran parte delle applicazioni, e non aderirvi potrebbe portare a gravi problemi di interoperabilità.

Il primo file è sempre associato al cosiddetto *standard input*; è cioè il file da cui il processo si aspetta di ricevere i dati in ingresso. Il secondo file è il cosiddetto *standard output*, cioè quello su cui ci si aspetta debbano essere inviati i dati in uscita. Il terzo è lo *standard error*, su cui viene inviata l'uscita relativa agli errori. Nel caso della shell tutti questi file sono associati al terminale

---

<sup>1</sup>nel 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.

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. 6.1.

Costante	Significato
STDIN_FILENO	<i>file descriptor</i> dello <i>standard input</i>
STDOUT_FILENO	<i>file descriptor</i> dello <i>standard output</i>
STDERR_FILENO	<i>file descriptor</i> dello <i>standard error</i>

**Tabella 6.1:** Costanti definite in `unistd.h` per i file standard aperti alla creazione di ogni processo.

In fig. 6.1 si è rappresentata una situazione diversa, facendo riferimento ad un programma in cui lo *standard input* è associato ad un file mentre lo *standard output* e lo *standard error* sono entrambi associati ad un altro file (e quindi utilizzano lo stesso 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 `file_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. 8.1.1).

## 6.2 Le funzioni base

L'interfaccia standard Unix per l'input/output sui file è basata su cinque funzioni fondamentali: `open`, `read`, `write`, `lseek` e `close`, usate rispettivamente per aprire, leggere, scrivere, spostarsi e chiudere un file. La gran parte delle operazioni sui file si effettua attraverso queste cinque funzioni, esse vengono chiamate anche funzioni di I/O non bufferizzato dato che effettuano le operazioni di lettura e scrittura usando direttamente le system call del kernel.

### 6.2.1 La funzione open

La funzione `open` è la funzione fondamentale per accedere ai file, ed è quella che crea l'associazione fra un *pathname* ed un file descriptor, il suo prototipo è:

```
#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 il file indicato da `pathname` nella modalità indicata da `flags`, e, nel caso il file sia creato, con gli eventuali permessi specificati da `mode`.

La funzione ritorna il file descriptor in caso di successo e -1 in caso di errore. In questo caso la variabile `errno` assumerà uno dei valori:

EEXIST	<code>pathname</code> esiste e si è specificato <code>O_CREAT</code> e <code>O_EXCL</code> .
EISDIR	<code>pathname</code> indica una directory e si è tentato l'accesso in scrittura.
ENOTDIR	si è specificato <code>O_DIRECTORY</code> e <code>pathname</code> non è una directory.
ENXIO	si sono impostati <code>O_NOBLOCK</code> o <code>O_WRONLY</code> ed il file è una fifo che non viene letta da nessun processo o <code>pathname</code> è un file di dispositivo ma il dispositivo è assente.
ENODEV	<code>pathname</code> si riferisce a un file di dispositivo che non esiste.
ETXTBSY	si è cercato di accedere in scrittura all'immagine di un programma in esecuzione.
ELOOP	si sono incontrati troppi link simbolici nel risolvere il <code>pathname</code> o si è indicato <code>O_NOFOLLOW</code> e <code>pathname</code> è un link simbolico.

ed inoltre `EACCES`, `ENAMETOOLONG`, `ENOENT`, `EROFS`, `EFAULT`, `ENOSPC`, `ENOMEM`, `EMFILE` e `ENFILE`.

La funzione apre il file, usando il primo file descriptor libero, e crea l'opportuna voce (cioè la struttura `file`) nella *file table*. Viene sempre restituito come valore di ritorno il file descriptor con il valore più basso disponibile.

Flag	Descrizione
<code>O_RDONLY</code>	apre il file in sola lettura, le <i>glibc</i> definiscono anche <code>O_READ</code> come sinonimo.
<code>O_WRONLY</code>	apre il file in sola scrittura, le <i>glibc</i> definiscono anche <code>O_WRITE</code> come sinonimo.
<code>O_RDWR</code>	apre il file sia in lettura che in scrittura.
<code>O_CREAT</code>	se il file non esiste verrà creato, con le regole di titolarità del file viste in sez. 5.3.4. Con questa opzione l'argomento <code>mode</code> deve essere specificato.
<code>O_EXCL</code>	usato in congiunzione con <code>O_CREAT</code> fa sì che la precedente esistenza del file diventi un errore <sup>2</sup> che fa fallire <code>open</code> con <code>EEXIST</code> .
<code>O_NONBLOCK</code>	apre il file in modalità non bloccante. Questo valore specifica anche una modalità di operazione (vedi sotto), e comporta che <code>open</code> ritorni immediatamente (l'opzione ha senso solo per le fifo, torneremo questo in sez. 12.1.4).
<code>O_NOCTTY</code>	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. 10.1.3).
<code>O_SHLOCK</code>	opzione di BSD, acquisisce uno shared lock (vedi sez. 11.4) sul file. Non è disponibile in Linux.
<code>O_EXLOCK</code>	opzione di BSD, acquisisce uno lock esclusivo (vedi sez. 11.4) sul file. Non è disponibile in Linux.
<code>O_TRUNC</code>	se il file esiste ed è un file di dati e la modalità di apertura consente la scrittura, allora la sua lunghezza verrà troncata a zero. Se il file è un terminale o una fifo il flag verrà ignorato, negli altri casi il comportamento non è specificato.
<code>O_NOFOLLOW</code>	se <code>pathname</code> è un link simbolico la chiamata fallisce. Questa è un'estensione BSD aggiunta in Linux dal kernel 2.1.126. Nelle versioni precedenti i link simbolici sono sempre seguiti, e questa opzione è ignorata.
<code>O_DIRECTORY</code>	se <code>pathname</code> non è una directory la chiamata fallisce. Questo flag è specifico di Linux ed è stato introdotto con il kernel 2.1.126 per evitare dei <i>DoS</i> <sup>3</sup> quando <code>opendir</code> viene chiamata su una fifo o su un device di unità a nastri, non deve essere utilizzato al di fuori dell'implementazione di <code>opendir</code> .
<code>O_LARGEFILE</code>	nel caso di sistemi a 32 bit che supportano file di grandi dimensioni consente di aprire file le cui dimensioni non possono essere rappresentate da numeri a 31 bit.
<code>O_APPEND</code>	il file viene aperto in append mode. Prima di ciascuna scrittura la posizione corrente viene sempre impostata alla fine del file. Con NFS si può avere una corruzione del file se più di un processo scrive allo stesso tempo. <sup>4</sup>
<code>O_NONBLOCK</code>	il file viene aperto in modalità non bloccante per le operazioni di I/O (che tratteremo in sez. 11.1.1): questo significa il fallimento di <code>read</code> in assenza di dati da leggere e quello di <code>write</code> in caso di impossibilità di scrivere immediatamente. Questa modalità ha senso solo per le fifo e per alcuni file di dispositivo.
<code>O_NDELAY</code>	in Linux <sup>5</sup> è sinonimo di <code>O_NONBLOCK</code> .
<code>O_ASYNC</code>	apre il file per l'I/O in modalità asincrona (vedi sez. 11.2.2). Quando è impostato viene generato il segnale <code>SIGIO</code> tutte le volte che sono disponibili dati in input sul file.
<code>O_SYNC</code>	apre il file per l'input/output sincrono: ogni <code>write</code> bloccherà fino al completamento della scrittura di tutti i dati sull'hardware sottostante.
<code>O_FSYNC</code>	sinonimo di <code>O_SYNC</code> , usato da BSD.
<code>O_DSYNC</code>	richiede una variante di I/O sincrono definita nello standard POSIX; definita a partire dal kernel 2.1.130 come sinonimo di <code>O_SYNC</code> .
<code>O_RSYNC</code>	richiede una variante di I/O sincrono definita nello standard POSIX; definita a partire dal kernel 2.1.130 come sinonimo di <code>O_SYNC</code> .
<code>O_NOATIME</code>	blocca l'aggiornamento dei tempi di accesso dei file (vedi sez. 5.2.4). Per molti filesystem questa funzionalità non è disponibile per il singolo file ma come opzione generale da specificare in fase di montaggio.
<code>O_DIRECT</code>	esegue l'I/O direttamente dai buffer in user space, ed in maniera sincrona, in modo da scavalcare i meccanismi di caching del kernel. In genere questo peggiora le prestazioni tranne per casi speciali in cui sono le applicazioni <sup>6</sup> a gestire il caching. Per i kernel della serie 2.4 si deve garantire che i buffer in user space siano allineati alle dimensioni dei blocchi del filesystem; per il kernel 2.6 basta che siano allineati a multipli di 512 byte.

**Tabella 6.2:** Valori e significato dei vari bit del *file status flag*.

Questa caratteristica permette di prevedere qual'è il valore del file descriptor che si otterrà al ritorno di `open`, e viene talvolta usata da alcune applicazioni per sostituire i file corrispondenti ai file standard visti in sez. 6.1.2: se ad esempio si chiude lo standard input e si apre subito dopo un nuovo file questo diventerà il nuovo standard input (avrà cioè il file descriptor 0).

Il nuovo file descriptor non è condiviso con nessun altro processo (torneremo sulla condivisione dei file, in genere accessibile dopo una `fork`, in sez. 6.3.1) ed è impostato per restare aperto attraverso una `exec` (come accennato in sez. 3.2.7); l'offset è impostato all'inizio del file.

L'argomento `mode` indica i permessi con cui il file viene creato; i valori possibili sono gli stessi già visti in sez. 5.3.1 e possono essere specificati come OR binario delle costanti descritte in tab. 5.7. Questi permessi sono filtrati dal valore di `umask` (vedi sez. 5.3.7) per il processo.

La funzione prevede diverse opzioni, che vengono specificate usando vari bit dell'argomento `flags`. Alcuni di questi bit vanno anche a costituire il flag di stato del file (o *file status flag*), che è mantenuto nel campo `f_flags` della struttura `file` (al solito si veda lo schema di fig. 6.1). Essi sono divisi in tre categorie principali:

- *i bit delle modalità di accesso*: specificano con quale modalità si accederà al file: i valori possibili sono lettura, scrittura o lettura/scrittura. Uno di questi bit deve essere sempre specificato quando si apre un file. Vengono impostati alla chiamata da `open`, e possono essere riletti con `fcntl` (fanno parte del *file status flag*), ma non possono essere modificati.
- *i bit delle modalità di apertura*: permettono di specificare alcune delle caratteristiche del comportamento di `open` quando viene eseguita. Hanno effetto solo al momento della chiamata della funzione e non sono memorizzati né possono essere riletti.
- *i bit delle modalità di operazione*: permettono di specificare alcune caratteristiche del comportamento delle future operazioni sul file (come `read` o `write`). Anch'essi fan parte del *file status flag*. Il loro valore è impostato alla chiamata di `open`, ma possono essere riletti e modificati (insieme alle caratteristiche operative che controllano) con una `fcntl`.

In tab. 6.2 sono riportate, ordinate e divise fra loro secondo le tre modalità appena elencate, le costanti mnemoniche associate a ciascuno di questi bit. Dette costanti possono essere combinate fra loro con un OR aritmetico per costruire il valore (in forma di maschera binaria) dell'argomento `flags` da passare alla `open`. I due flag `O_NOFOLLOW` e `O_DIRECTORY` sono estensioni specifiche di Linux, e deve essere definita la macro `_GNU_SOURCE` per poterli usare.

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`, il cui prototipo è:

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode)
    Crea un nuovo file vuoto, con i permessi specificati da mode. È del tutto equivalente a
    open(filedes, O_CREAT|O_WRONLY|O_TRUNC, mode).
```

adesso questa funzione resta solo per compatibilità con i vecchi programmi.

<sup>2</sup>la pagina di manuale di `open` segnala che questa opzione è difettosa su NFS, e che i programmi che la usano per stabilire un *file di lock* possono incorrere in una race condition. Si consiglia come alternativa di usare un file con un nome univoco e la funzione `link` per verificarne l'esistenza (vedi sez. 12.3.2).

<sup>3</sup>*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.

<sup>4</sup>il problema è che NFS non supporta la scrittura in append, ed il kernel deve simularla, ma questo comporta la possibilità di una race condition, vedi sez. 6.3.2.

<sup>5</sup>l'opzione origina da SVr4, dove però causava il ritorno da una `read` con un valore nullo e non con un errore, questo introduce un'ambiguità, dato che come vedremo in sez. 6.2.4 il ritorno di zero da parte di `read` ha il significato di una end-of-file.

<sup>6</sup>l'opzione è stata introdotta dalla SGI in IRIX, e serve sostanzialmente a permettere ad alcuni programmi (in genere database) la gestione diretta della bufferizzazione dell'I/O in quanto essi sono in grado di ottimizzarla al meglio per le loro prestazioni; l'opzione è presente anche in FreeBSD, senza limiti di allineamento dei buffer. In Linux è stata introdotta con il kernel 2.4.10, le versioni precedenti la ignorano.

### 6.2.2 La funzione close

La funzione `close` permette di chiudere un file, in questo modo il file descriptor ritorna disponibile; il suo prototipo è:

```
#include <unistd.h>
int close(int fd)
    Chiude il descrittore fd.
```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, con `errno` che assume i valori:

<b>EBADF</b>	<code>fd</code> non è un descrittore valido.
<b>EINTR</b>	la funzione è stata interrotta da un segnale.
ed inoltre <b>EIO</b> .	

La chiusura di un file rilascia ogni blocco (il *file locking* è trattato in sez. 11.4) che il processo poteva avere acquisito su di esso; se `fd` è l'ultimo riferimento (di eventuali copie) 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 anche tutti i suoi file descriptor vengono chiusi, molti programmi sfruttano questa caratteristica e non usano esplicitamente `close`. In genere comunque chiudere un file senza controllarne lo stato di uscita è errore; infatti molti filesystem implementano la tecnica del *write-behind*, per cui una `write` può avere successo anche se i dati non sono stati scritti, un eventuale errore di I/O allora può sfuggire, ma verrà riportato alla chiusura del file: per questo motivo non effettuare il controllo può portare ad una perdita di dati inavvertita.<sup>7</sup>

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. 6.3.3) effettua esplicitamente il *flush* 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 cui l'abitudine di ripetere tre volte il comando prima di eseguire lo shutdown).

### 6.2.3 La funzione lseek

Come già accennato in sez. 6.1.1 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 come 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 `O_APPEND`) questa posizione viene impostata a zero all'apertura del file. È possibile impostarla ad un valore qualsiasi con la funzione `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 attuale nel file.
```

La funzione ritorna il valore della posizione corrente in caso di successo e -1 in caso di errore nel qual caso `errno` assumerà uno dei valori:

<b>ESPIPE</b>	<code>fd</code> è una pipe, un socket o una fifo.
<b>EINVAL</b>	<code>whence</code> non è un valore valido.
ed inoltre <b>EBADF</b> .	

<sup>7</sup>in Linux questo comportamento è stato osservato con NFS e le quote su disco.

La nuova posizione è impostata usando il valore specificato da `offset`, sommato al riferimento dato da `whence`; quest'ultimo può assumere i seguenti valori<sup>8</sup>:

- `SEEK_SET` si fa riferimento all'inizio del file: il valore (sempre positivo) di `offset` indica direttamente la nuova posizione corrente.
- `SEEK_CUR` si fa riferimento alla posizione corrente del file: ad essa viene sommato `offset` (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 `offset` (che può essere negativo e positivo) per ottenere la nuova posizione corrente.

Come accennato in sez. 5.2.3 con `lseek` è possibile impostare la posizione corrente anche oltre la fine del file, e alla successiva scrittura il file sarà esteso. La chiamata non causa nessun accesso al file, si limita a modificare la posizione corrente (cioè il valore `f_pos` in `file`, vedi fig. 6.1). Dato che la funzione ritorna la nuova posizione, usando il valore zero per `offset` si può riottenerne la posizione corrente nel file chiamando la funzione 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 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. 6.3.2).

Non tutti i file supportano la capacità di eseguire una `lseek`, in questo caso la funzione ritorna l'errore `EPIPE`. 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.<sup>9</sup> Lo standard POSIX però non specifica niente in proposito. Infine alcuni file speciali, ad esempio `/dev/null`, non causano un errore ma restituiscono un valore indefinito.

#### 6.2.4 La funzione `read`

Una volta che un file è stato aperto (con il permesso in lettura) si possono leggere i dati che contiene utilizzando la funzione `read`, il cui prototipo è:

```
#include <unistd.h>
ssize_t read(int fd, void * buf, size_t count)
    Cerca di leggere count byte dal file fd al buffer buf.
```

La funzione ritorna il numero di byte letti in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

- `EINTR` la funzione è stata interrotta da un segnale prima di aver potuto leggere qualsiasi dato.
- `EAGAIN` la funzione non aveva nessun dato da restituire e si era aperto il file in modalità `O_NONBLOCK`.

ed inoltre `EBADF`, `EIO`, `EISDIR`, `EBADF`, `EINVAL` e `EFAULT` ed eventuali altri errori dipendenti dalla natura dell'oggetto connesso a `fd`.

La funzione tenta di leggere `count` byte a partire dalla posizione corrente nel file. 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. Si deve sempre tener presente che non è detto che la funzione `read` restituisca sempre 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 regolarmente il

<sup>8</sup>per 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`.

<sup>9</sup>altri sistemi, usando `SEEK_SET`, in questo caso ritornano il numero di caratteri che vi sono stati scritti.

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. 11.1.1) 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.

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. 9.3.1. La seconda si verifica quando il file è aperto in modalità non bloccante (vedi sez. 11.1.1) e non ci sono dati in ingresso: la funzione allora ritorna immediatamente con un errore `EAGAIN`<sup>10</sup> che indica soltanto che non essendoci al momento dati disponibili occorre provare a ripetere la lettura in un secondo tempo.

La funzione `read` è una delle system call fondamentali, esistenti fin dagli albori di Unix, ma nella seconda versione delle *Single Unix Specification*<sup>11</sup> (quello che viene chiamato normalmente Unix98, vedi sez. 1.2.5) è stata introdotta la definizione di un'altra funzione di lettura, `pread`, il cui prototipo è:

```
#include <unistd.h>
ssize_t pread(int fd, void * buf, size_t count, off_t offset)
    Cerca di leggere count byte dal file fd, a partire dalla posizione offset, nel buffer buf.
```

La funzione ritorna il numero di byte letti in caso di successo e -1 in caso di 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. Indetico è il comportamento ed il valore di ritorno. La funzione serve quando si vogliono leggere dati dal file senza modificare la posizione corrente.

L'uso di `pread` è equivalente all'esecuzione di una `read` seguita da una `lseek` che riporti al valore precedente la 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. 6.3.1). Il valore di `offset` fa sempre riferimento all'inizio del file.

La funzione `pread` è disponibile anche in Linux, però diventa accessibile solo attivando il supporto delle estensioni previste dalle *Single Unix Specification* con la definizione della macro:

```
#define _XOPEN_SOURCE 500
```

e si ricordi di definire questa macro prima dell'inclusione del file di dichiarazioni `unistd.h`.

<sup>10</sup>BSD usa per questo errore la costante `EWOULDBLOCK`, in Linux, con le *glibc*, questa è sinonima di `EAGAIN`.

<sup>11</sup>questa funzione, e l'analoga `pwrite` sono state aggiunte nel kernel 2.1.60, il supporto nelle *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.

### 6.2.5 La funzione write

Una volta che un file è stato aperto (con il permesso in scrittura) si può scrivere su di esso utilizzando la funzione `write`, il cui prototipo è:

```
#include <unistd.h>
ssize_t write(int fd, void * buf, size_t count)
    Scrive count byte dal buffer buf sul file fd.
```

La funzione ritorna il numero di byte scritti in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EINVAL</code>	<code>fd</code> è connesso ad un oggetto che non consente la scrittura.
<code>EFBIG</code>	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.
<code>EPIPE</code>	<code>fd</code> è connesso ad una pipe il cui altro capo è chiuso in lettura; in questo caso viene anche generato il segnale <code>SIGPIPE</code> , se questo viene gestito (o bloccato o ignorato) la funzione ritorna questo errore.
<code>EINTR</code>	si è stati interrotti da un segnale prima di aver potuto scrivere qualsiasi dato.
<code>EAGAIN</code>	ci si sarebbe bloccati, ma il file era aperto in modalità <code>O_NONBLOCK</code> .

ed inoltre `EBADF`, `EIO`, `EISDIR`, `EBADF`, `ENOSPC`, `EINVAL` e `EFAULT` ed eventuali altri errori dipendenti dalla natura dell'oggetto connesso a `fd`.

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 `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 definisce 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)
    Cerca di scrivere sul file fd, a partire dalla posizione offset, count byte dal buffer buf.
```

La funzione ritorna il numero di byte letti in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori già visti per `write` e `lseek`.

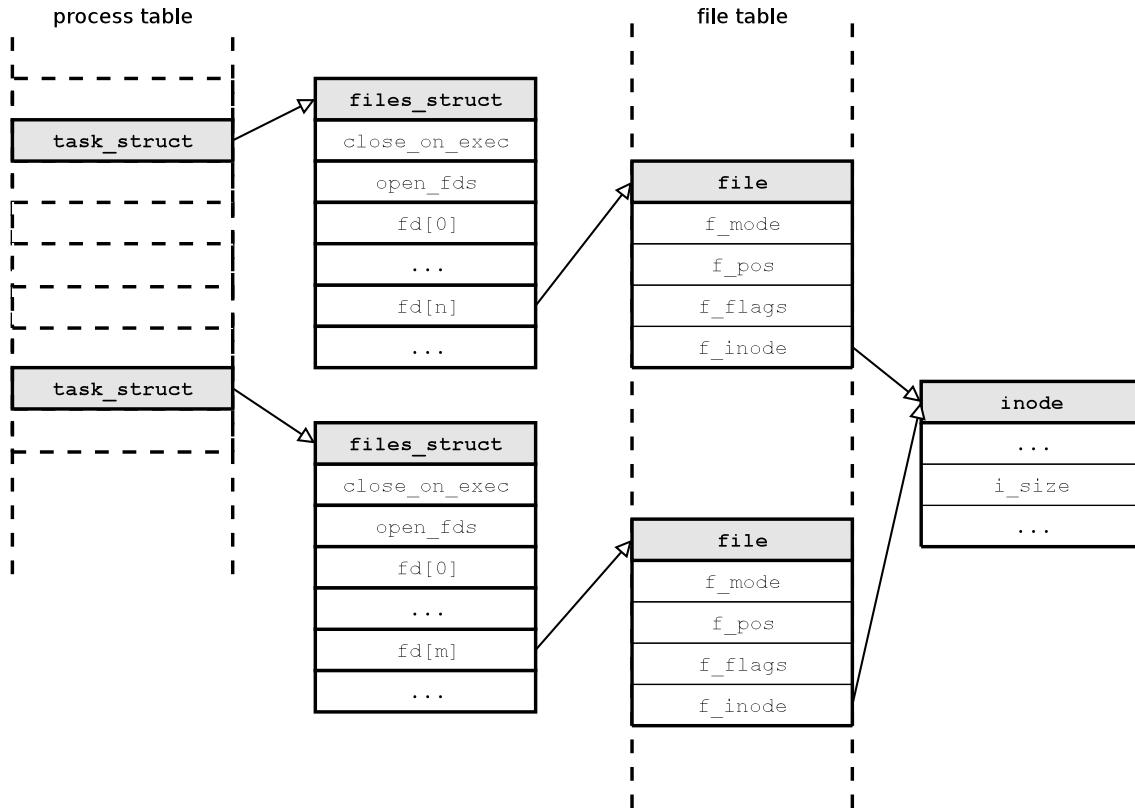
e per essa valgono le stesse considerazioni fatte per `pread`.

## 6.3 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 in cap. 11).

### 6.3.1 La condivisione dei files

In sez. 6.1.1 abbiamo descritto brevemente l'architettura dell'interfaccia con i file da parte di un processo, mostrando in fig. 6.1 le principali strutture usate dal kernel; esamineremo ora in dettaglio le conseguenze che questa architettura ha nei confronti dell'accesso allo stesso file da parte di processi diversi.



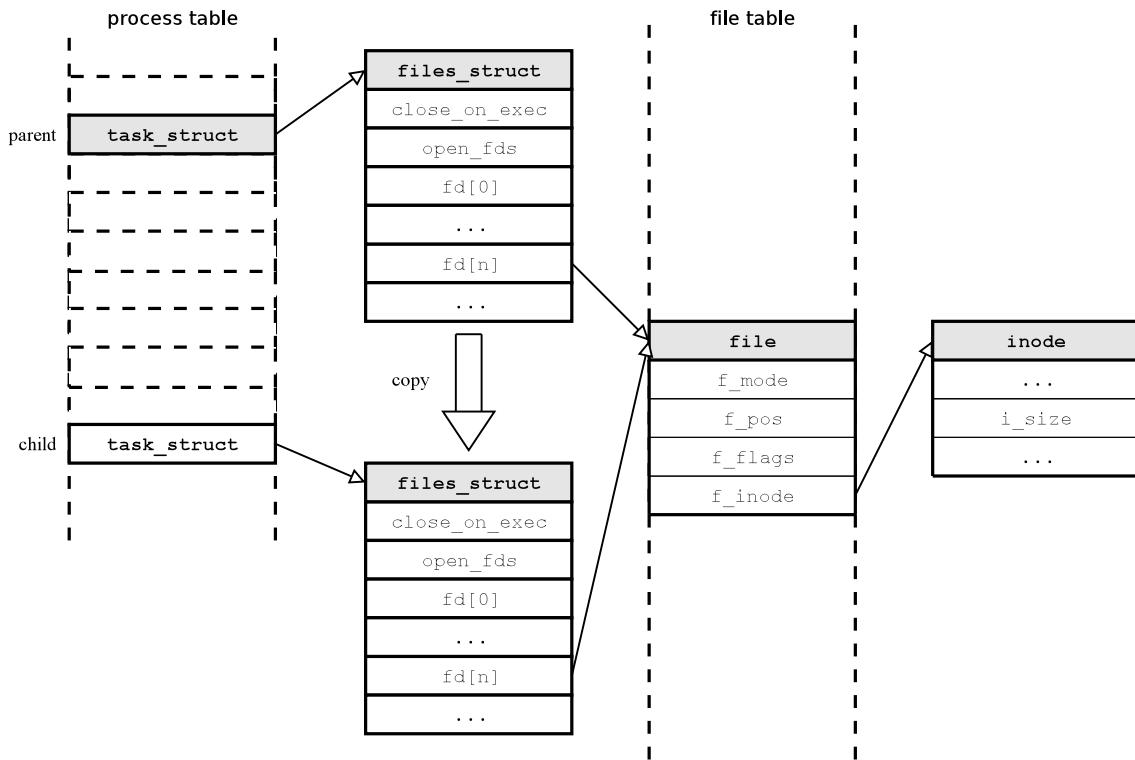
**Figura 6.2:** Schema dell'accesso allo stesso file da parte di due processi diversi

Il primo caso è quello in cui due processi diversi aprono lo stesso file su disco; sulla base di quanto visto in sez. 6.1.1 avremo una situazione come quella illustrata in fig. 6.2: 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. Se la scrittura eccede la dimensione corrente del file questo verrà esteso automaticamente con l'aggiornamento del campo `i_size` nell'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 dall'inode. Dopo la scrittura il file viene automaticamente esteso.
- 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 dimensione corrente dall'inode.

Il secondo caso è quello in cui due file descriptor di due processi diversi puntino 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. 3.2.2). La situazione è illustrata



**Figura 6.3:** Schema dell'accesso ai file da parte di un processo figlio

in fig. 6.3; dato che il processo figlio riceve una copia dello spazio di indirizzi del padre, riceverà anche una copia di **file\_struct** e relativa tabella dei file aperti.

In questo modo padre e figlio avranno gli stessi file descriptor che 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. 3.2.2: in caso di scrittura contemporanea la posizione corrente nel file varierà per entrambi i processi (in quanto verrà modificato **f\_pos** che è lo stesso per entrambi).

Si noti inoltre che anche i flag di stato del file (quelli impostati dall'argomento **flag** di **open**) essendo tenuti nella voce della *file table*<sup>12</sup>, vengono in questo caso condivisi. Ai file però sono associati anche altri flag, dei quali l'unico usato al momento è **FD\_CLOEXEC**, detti *file descriptor flags*. Questi ultimi sono tenuti invece in **file\_struct**, e perciò sono specifici di ciascun processo e non vengono modificati dalle azioni degli altri anche in caso di condivisione della stessa voce della *file table*.

### 6.3.2 Operazioni atomiche con i file

Come si è visto in un sistema unix-like è sempre possibile per più processi accedere in contemporanea allo stesso file, e che le operazioni di lettura e scrittura possono essere fatte da ogni processo in maniera autonoma in base ad una posizione corrente nel file che è locale a ciascuno di essi.

Se dal punto di vista della lettura dei dati questo non comporta nessun problema, quando si andrà a scrivere le operazioni potranno mescolarsi in maniera imprevedibile. Il sistema però fornisce in alcuni casi la possibilità di eseguire alcune operazioni di scrittura in maniera coordinata anche senza utilizzare meccanismi di sincronizzazione più complessi (come il *file locking*, che esamineremo in sez. 11.4).

<sup>12</sup>per la precisione nel campo **f\_flags** di **file**.

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. 6.2.3 impostare la posizione alla fine del file e poi scrivere 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 nostro primo processo avrà ancora la posizione corrente impostata con la `lseek` che non corrisponde più alla fine del file, e la successiva `write` sovrascriverà i dati del secondo processo.

Il problema è che usare due system call in successione non è un'operazione atomica; il problema è stato risolto introducendo la modalità `O_APPEND`. In questo caso infatti, come abbiamo descritto in precedenza, 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 costituisce un'operazione atomica.

Un altro caso tipico in cui è necessaria l'atomicità è quello in cui si vuole creare un *file di lock*, bloccandosi se il file esiste. In questo caso la sequenza logica porterebbe a verificare prima l'esistenza del file con una `stat` per poi crearlo con una `creat`; di nuovo avremmo la possibilità di una race condition da parte di un altro processo che crea lo stesso file fra il controllo e la creazione.

Per questo motivo sono stati introdotti per `open` i due flag `O_CREAT` e `O_EXCL`. In questo modo l'operazione di controllo dell'esistenza del file (con relativa uscita dalla funzione con un errore) e creazione in caso di assenza, diventa atomica essendo svolta tutta all'interno di una singola system call (per i dettagli sull'uso di questa caratteristica si veda sez. 12.3.2).

### 6.3.3 La funzioni sync e fsync

Come accennato in sez. 6.2.2 tutte le operazioni di scrittura sono in genere bufferizzate dal kernel, che provvede ad effettuarle in maniera asincrona (ad esempio accorpando 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 dei dati, il sistema mette a disposizione delle funzioni che provvedono a forzare lo scarico dei dati dai buffer del kernel.<sup>13</sup> La prima di queste funzioni è `sync` il cui prototipo è:

```
#include <unistd.h>
int sync(void)
    Sincronizza il buffer della cache dei file col disco.

La funzione ritorna sempre zero.
```

i vari standard prevedono che la funzione si limiti a far partire le operazioni, ritornando immediatamente; in Linux (dal kernel 1.3.20) invece la funzione aspetta la conclusione delle operazioni di sincronizzazione del kernel.

La funzione viene usata dal comando `sync` quando si vuole forzare esplicitamente lo scarico dei dati su disco, o dal demone di sistema `update` che esegue lo scarico dei dati ad intervalli di tempo fissi: il valore tradizionale, usato da BSD, per l'update dei dati è ogni 30 secondi, ma in Linux il valore utilizzato è di 5 secondi; con le nuove versioni<sup>14</sup> poi, è il kernel che si occupa direttamente di tutto quanto attraverso il demone interno `bdflush`, il cui comportamento può essere controllato attraverso il file `/proc/sys/vm/bdflush` (per il significato dei valori si può leggere la documentazione allegata al kernel in `Documentation/sysctl/vm.txt`).

<sup>13</sup>come già accennato neanche questo 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 ottimizzazione per l'accesso al disco che può ritardare ulteriormente la scrittura effettiva.

<sup>14</sup>a partire dal kernel 2.2.8

Quando si vogliono scaricare soltanto i dati di un file (ad esempio essere sicuri che i dati di un database sono stati registrati su disco) si possono usare le due funzioni `fsync` e `fdatasync`, i cui prototipi sono:

```
#include <unistd.h>
int fsync(int fd)
    Sincronizza dati e metadati del file fd
int fdatasync(int fd)
    Sincronizza i dati del file fd.
```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assume i valori:

`EINVAL`     `fd` è un file speciale che non supporta la sincronizzazione.  
ed inoltre `EBADF`, `EROFS` e `EIO`.

Entrambe le funzioni forzano la sincronizzazione col disco di tutti i dati del file specificato, ed attendono fino alla conclusione delle operazioni; `fsync` forza anche la sincronizzazione dei metadati 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).

Si tenga presente che questo non comporta la sincronizzazione della directory che contiene il file (e scrittura della relativa voce su disco) che deve essere effettuata esplicitamente.<sup>15</sup>

### 6.3.4 La funzioni `dup` e `dup2`

Abbiamo già visto in sez. 6.3.1 come un processo figlio condivida gli stessi file descriptor del padre; è possibile però ottenere un comportamento analogo all'interno di uno stesso processo *duplicando* un file descriptor. Per far questo si usa la funzione `dup` il cui prototipo è:

```
#include <unistd.h>
int dup(int oldfd)
    Crea una copia del file descriptor oldfd.
```

La funzione ritorna il nuovo file descriptor in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

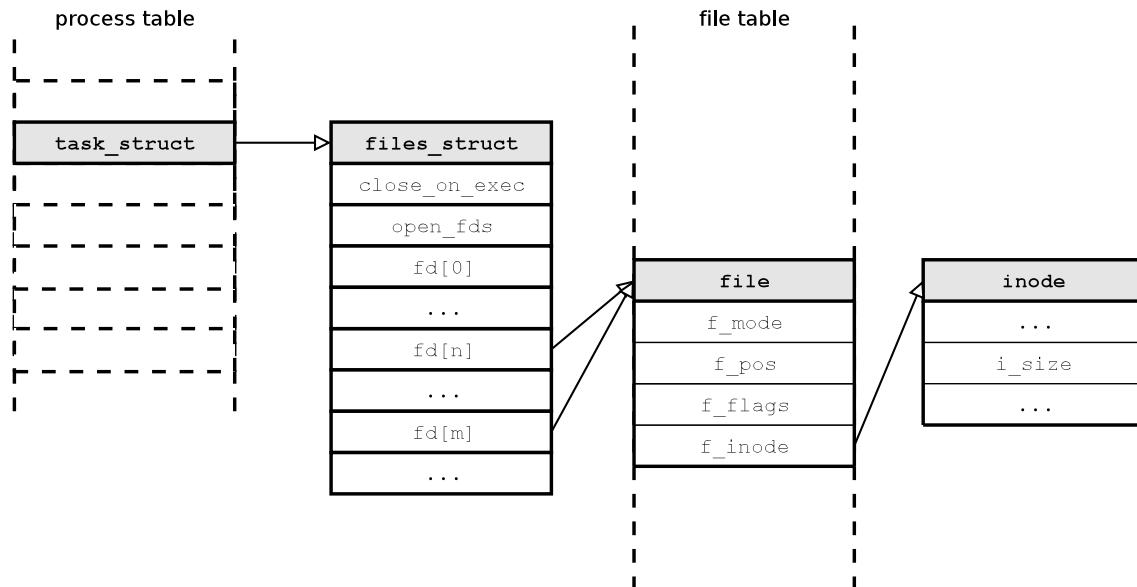
`EBADF`     `oldfd` non è un file aperto.  
`EMFILE`     si è raggiunto il numero massimo consentito di file descriptor aperti.

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. 6.4: l'effetto della funzione è semplicemente quello di copiare il valore nella struttura `file_struct`, cosicché anche il nuovo file descriptor fa riferimento alla stessa voce nella *file table*; per questo si dice che il nuovo file descriptor è *duplicato*, da cui il nome della funzione.

Si noti che per quanto illustrato in fig. 6.4 i file descriptor duplicati condivideranno eventuali lock, *file status flag*, e posizione corrente. 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 il suo *file descriptor flag*; a questo proposito va specificato che nel caso di `dup` il flag di *close-on-exec* (vedi sez. 3.2.7 e sez. 6.3.5) viene sempre cancellato nella copia.

L'uso principale di questa funzione è per la redirezione dell'input e dell'output 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 (torneremo sull'argomento in sez. 12.1.2, quando tratteremo le pipe). Per fare questo in genere occorre prima chiudere il file che si vuole sostituire, cosicché

<sup>15</sup>in realtà per il filesystem `ext2`, quando lo si monta con l'opzione `sync`, il kernel provvede anche alla sincronizzazione automatica delle voci delle directory.



**Figura 6.4:** Schema dell'accesso ai file duplicati

il suo file descriptor possa esser restituito alla chiamata di `dup`, come primo file descriptor disponibile.

Dato che questa è l'operazione più comune, è prevista una diversa versione della funzione, `dup2`, che permette di specificare esplicitamente qual'è il valore di file descriptor che si vuole avere come duplicato; il suo prototipo è:

```
#include <unistd.h>
int dup2(int oldfd, int newfd)
    Rende newfd una copia del file descriptor oldfd.

La funzione ritorna il nuovo file descriptor in caso di successo e -1 in caso di 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.
EMFILE   si è raggiunto il numero massimo consentito di file descriptor aperti.
```

e qualora il file descriptor `newfd` sia già aperto (come avviene ad esempio nel caso della duplicazione di uno dei file standard) esso sarà prima chiuso e poi duplicato (così che il file duplicato sarà connesso allo stesso valore per il file descriptor).

La duplicazione dei file descriptor può essere effettuata anche usando la funzione di controllo dei file `fcntl` (che esamineremo in sez. 6.3.5) 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<sup>16</sup> 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` (e se `newfd` è aperto la duplicazione avverrà su un altro file descriptor).

<sup>16</sup>a parte la sintassi ed i diversi codici di errore.

### 6.3.5 La funzione fcntl

Oltre alle operazioni base esaminate in sez. 6.2 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.<sup>17</sup>

Per queste operazioni di manipolazione e di controllo delle varie proprietà e caratteristiche di un file descriptor, viene usata la funzione `fcntl`, il cui prototipo è:

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd)
int fcntl(int fd, int cmd, long arg)
int fcntl(int fd, int cmd, struct flock * lock)
```

Esegue una delle possibili operazioni specificate da `cmd` sul file `fd`.

La funzione ha valori di ritorno diversi a seconda dell'operazione. In caso di errore il valore di ritorno è sempre -1 ed il codice dell'errore è restituito nella variabile `errno`; i codici possibili dipendono dal tipo di operazione, l'unico valido in generale è:

**EBADF**      `fd` non è un file aperto.

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 sono determinati dal valore dell'argomento `cmd` che in sostanza corrisponde all'esecuzione di un determinato *comando*; in sez. 6.3.4 abbiamo incontrato un esempio dell'uso di `fcntl` per la duplicazione dei file descriptor, una lista di tutti i possibili valori per `cmd` è riportata di seguito:

- F\_DUPFD**      trova il primo file descriptor disponibile di valore maggiore o uguale ad `arg` e ne fa una copia di `fd`. Ritorna il nuovo file descriptor in caso di successo e -1 in caso di errore. 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\_SETFD**      imposta il valore del *file descriptor flag* al valore specificato con `arg`. Al momento l'unico bit 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. 3.2.7). Ritorna un valore nullo in caso di successo e -1 in caso di errore.
- F\_GETFD**      ritorna il valore del *file descriptor flag* di `fd` o -1 in caso di errore; se `FD_CLOEXEC` è impostato i file descriptor aperti vengono chiusi attraverso una `exec` altrimenti (il comportamento predefinito) restano aperti.
- F\_GETFL**      ritorna il valore del *file status flag* in caso di successo o -1 in caso di errore; permette cioè di rileggere quei bit impostati da `open` all'apertura del file che vengono memorizzati (quelli riportati nella prima e terza sezione di tab. 6.2).
- F\_SETFL**      imposta il *file status flag* al valore specificato da `arg`, ritorna un valore nullo in caso di successo o -1 in caso di errore. Possono essere impostati solo i bit riportati nella terza sezione di tab. 6.2.<sup>18</sup>
- F\_GETLK**      richiede un controllo sul file lock specificato da `lock`, sovrascrivendo la struttura da esso puntata con il risultato, ritorna un valore nullo in caso di successo o -1 in caso di errore. Questa funzionalità è trattata in dettaglio in sez. 11.4.3.

<sup>17</sup>ad esempio si gestiscono con questa funzione varie modalità di I/O asincrono (vedi sez. 11.2.1) e il file locking (vedi sez. 11.4).

<sup>18</sup>la pagina di manuale riporta come impostabili solo `O_APPEND`, `O_NONBLOCK` e `O_ASYNC`.

F_SETLK	richiede o rilascia un file lock a seconda di quanto specificato nella struttura puntata da <code>lock</code> . Se il lock è tenuto da qualcun’altro ritorna immediatamente restituendo -1 e imposta <code>errno</code> a <code>EACCES</code> o <code>EAGAIN</code> , in caso di successo ritorna un valore nullo. Questa funzionalità è trattata in dettaglio in sez. 11.4.3.
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 <code>errno</code> a <code>EINTR</code> , in caso di successo ritorna un valore nullo. Questa funzionalità è trattata in dettaglio in sez. 11.4.3.
F_GETOWN	restituisce il <i>pid</i> del processo o l’identificatore del process group <sup>19</sup> che è preposto alla ricezione dei segnali <code>SIGIO</code> e <code>SIGURG</code> per gli eventi associati al file descriptor <code>fd</code> . Nel caso di un process group viene restituito un valore negativo il cui valore assoluto corrisponde all’identificatore del process group. In caso di errore viene restituito -1.
F_SETOWN	imposta, con il valore dell’argomento <code>arg</code> , l’identificatore del processo o del <i>process group</i> che riceverà i segnali <code>SIGIO</code> e <code>SIGURG</code> per gli eventi associati al file descriptor <code>fd</code> , ritorna un valore nullo in caso di successo o -1 in caso di errore. Come per <code>F_GETOWN</code> , per impostare un <i>process group</i> si deve usare per <code>arg</code> un valore negativo, il cui valore assoluto corrisponde all’identificatore del <i>process group</i> .
F_GETSIG	restituisce il valore del segnale inviato quando ci sono dati disponibili in ingresso su un file descriptor aperto ed impostato per l’I/O asincrono (si veda sez. 11.2.2). Il valore 0 indica il valore predefinito (che è <code>SIGIO</code> ), un valore diverso da zero indica il segnale richiesto, (che può essere anche lo stesso <code>SIGIO</code> ). In caso di errore ritorna -1.
F_SETSIG	imposta il segnale da inviare quando diventa possibile effettuare I/O sul file descriptor in caso di I/O asincrono, ritorna un valore nullo in caso di successo o -1 in caso di errore. Il valore zero indica di usare il segnale predefinito, <code>SIGIO</code> . Un altro valore diverso da zero (compreso lo stesso <code>SIGIO</code> ) specifica il segnale voluto; l’uso di un valore diverso da zero permette inoltre, se si è installato il gestore del segnale come <code>sa_sigaction</code> usando <code>SA_SIGINFO</code> , (vedi sez. 9.4.3), di rendere disponibili al gestore informazioni ulteriori riguardo il file che ha generato il segnale attraverso i valori restituiti in <code>siginfo_t</code> (come vedremo in sez. 11.2.2). <sup>20</sup>
F_SETLEASE	imposta o rimuove un <i>file lease</i> <sup>21</sup> sul file descriptor <code>fd</code> a seconda del valore del terzo argomento, che in questo caso è un <code>int</code> , ritorna un valore nullo in caso di successo o -1 in caso di errore. Questa funzionalità avanzata è trattata in dettaglio in sez. 11.2.1.
F_GETLEASE	restituisce il tipo di <i>file lease</i> che il processo detiene nei confronti del file descriptor <code>fd</code> o -1 in caso di errore. Con questo comando il terzo argomento può essere omesso. Questa funzionalità avanzata è trattata in dettaglio in sez. 11.2.1.
F_NOTIFY	attiva un meccanismo di notifica per cui viene riportata al processo chiamante, tramite il segnale <code>SIGIO</code> (o altro segnale specificato con <code>F_SETSIG</code> ) ogni modifica

<sup>19</sup>i `process group` sono (vedi sez. 10.1.2) raggruppamenti di processi usati nel controllo di sessione; a ciascuno di essi è associato un identificatore (un numero positivo analogo al *pid*).

<sup>20</sup>i due comandi `F_SETSIG` e `F_GETSIG` sono una estensione specifica di Linux.

<sup>21</sup>questa è una nuova funzionalità, specifica di Linux, e presente solo a partire dai kernel della serie 2.4.x, in cui il processo che detiene un *lease* su un file riceve una notifica qualora un altro processo cerca di eseguire una `open` o una `truncate` su di esso.

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. Questa funzionalità avanzata, disponibile dai kernel della serie 2.4.x, è trattata in dettaglio in sez. 11.2.1.

La maggior parte delle funzionalità di `fcntl` sono troppo avanzate per poter essere affrontate in tutti i loro aspetti a questo punto; 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. 11.2.1 mentre quelle relative al *file locking* saranno esaminate in sez. 11.4).

Si tenga presente infine che quando si usa la funzione per determinare le modalità di accesso con cui è stato aperto il file (attraverso l'uso del comando `F_GETFL`) è necessario estrarre i bit corrispondenti nel *file status flag* che si è ottenuto. Infatti la definizione corrente di quest'ultimo non assegna bit separati alle tre diverse modalità `O_RDONLY`, `O_WRONLY` e `O_RDWR`.<sup>22</sup> Per questo motivo il valore della modalità di accesso corrente si ottiene eseguendo un AND binario del valore di ritorno di `fcntl` con la maschera `O_ACCMODE` (anch'essa definita in `fcntl.h`), che estrae i bit di accesso dal *file status flag*.

### 6.3.6 La funzione `ioctl`

Benché il concetto di *everything is a file* si sia dimostrato molto valido anche per l'interazione con i dispositivi più vari, fornendo una interfaccia che permette di interagire con essi tramite le stesse funzioni usate per i normali file di dati, esisteranno sempre caratteristiche peculiari, specifiche dell'hardware e della funzionalità che ciascun dispositivo può provvedere, che non possono venire comprese in questa interfaccia astratta (un caso tipico è 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 funzione apposita, `ioctl`, con cui poter compiere le operazioni specifiche di ogni dispositivo particolare, usando come riferimento il solito file descriptor. Il prototipo di questa funzione è:

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ...)

Manipola il dispositivo sottostante, usando il parametro request per specificare l'operazione richiesta ed il terzo parametro (usualmente di tipo char * argp o int argp) per il trasferimento dell'informazione necessaria.

La funzione nella maggior parte dei casi ritorna 0, alcune operazioni usano però il valore di ritorno per restituire informazioni. In caso di errore viene sempre restituito -1 ed errno assumerà uno dei valori:
ENOTTY    il file fd non è associato con un device, o la richiesta non è applicabile all'oggetto a cui fa riferimento fd.
EINVAL    gli argomenti request o argp non sono validi.
ed inoltre EBADF e EFAULT.
```

La funzione serve in sostanza per fare tutte quelle operazioni che non si adattano al design dell'architettura dei file e che non è possibile effettuare con le funzioni esaminate finora. Esse vengono selezionate attraverso il valore di `request` e gli eventuali risultati possono essere restituiti sia attraverso il valore di ritorno che attraverso il terzo argomento `argp`. Sono esempi delle operazioni gestite con una `ioctl`:

- il cambiamento dei font di un terminale.
- l'esecuzione di una traccia audio di un CDROM.
- i comandi di avanti veloce e riavvolgimento di un nastro.

<sup>22</sup>in Linux queste costanti sono poste rispettivamente ai valori 0, 1 e 2.

- 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.

In generale ogni dispositivo ha un suo insieme di possibili diverse operazioni effettuabili attraverso `ioctl`, che 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 dispositivo<sup>23</sup> 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), qui riportiamo solo i valori di alcuni comandi che sono definiti per ogni file:

<code>FIOCLEX</code>	Imposta il bit di <i>close on exec</i> .
<code>FIONCLEX</code>	Cancella il bit di <i>close on exec</i> .
<code>FIOASYNC</code>	Abilita l'I/O asincrono.
<code>FIONBIO</code>	Abilita l'I/O in modalità non bloccante.

relativi ad operazioni comunque eseguibili anche attraverso `fcntl`.

---

<sup>23</sup>il 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 [4] per una trattazione dettagliata dell'argomento.

# Capitolo 7

## I file: l'interfaccia standard ANSI C

Esamineremo in questo capitolo l'interfaccia standard ANSI C per i file, quella che viene comunemente detta interfaccia degli *stream*. Dopo una breve sezione introduttiva tratteremo le funzioni base per la gestione dell'input/output, mentre tratteremo le caratteristiche più avanzate dell'interfaccia nell'ultima sezione.

### 7.1 Introduzione

Come visto in cap. 6 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, insieme alle altre funzioni definite dallo standard, vengono a costituire il nucleo<sup>1</sup> delle *glibc*.

#### 7.1.1 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 [1] Stevens descrive una serie di test sull'influenza delle dimensioni del blocco di dati (il parametro `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 della comunicazione con il tipo di hardware sottostante (come nel caso della dimensione dei blocchi del filesystem), ed un file può essere sempre considerato come composto da un flusso continuo (da cui il nome *stream*) di dati.

A parte i dettagli legati alla gestione delle operazioni di lettura e scrittura (sia per quel che riguarda la bufferizzazione, che le formattazioni), i file stream restano del tutto equivalenti ai file descriptor (sui quali sono basati), ed in particolare continua a valere quanto visto in sez. 6.3.1 a proposito dell'accesso condiviso ed in sez. 5.3 per il controllo di accesso.

---

<sup>1</sup>queste funzioni sono state implementate la prima volta da Ritchie nel 1976 e da allora sono rimaste sostanzialmente immutate.

### 7.1.2 Gli oggetti FILE

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 parametri solo variabili di questo tipo, che diventa accessibile includendo l'header file `stdio.h`.

### 7.1.3 Gli stream standard

Ai tre file descriptor standard (vedi sez. 6.1.2) 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 lo stream 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 lo stream 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 lo stream 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.

Nelle 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 redirezione 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`.

### 7.1.4 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 caratteri che vengono scritti su di uno stream normalmente vengono accumulati in un buffer e poi trasmessi in blocco<sup>2</sup> tutte le volte che il buffer viene riempito, 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

---

<sup>2</sup>questa operazione viene usualmente chiamata scaricamento dei dati, dal termine inglese *flush*.

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 (la cosa è particolarmente evidente quando con le operazioni di input/output su 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).
- *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 e 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* (*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 al kernel<sup>3</sup> viene anche eseguito lo scarico di tutti i buffer degli stream in scrittura.

In sez. 7.3.2 vedremo come la libreria definisca delle opportune funzioni per controllare le modalità di bufferizzazione e lo scarico dei dati.

## 7.2 Funzioni base

Esamineremo in questa sezione le funzioni base dell'interfaccia degli stream, analoghe a quelle di sez. 6.2 per i file descriptor. In particolare vedremo come aprire, leggere, scrivere e cambiare la posizione corrente in uno stream.

---

<sup>3</sup>questo vuol dire che lo stream da cui si legge è in modalità *unbuffered*.

### 7.2.1 Apertura e chiusura di uno stream

Le funzioni che si possono usare per aprire uno stream sono solo tre: `fopen`, `fdopen` e `freopen`,<sup>4</sup> i loro prototipi sono:

```
#include <stdio.h>
FILE *fopen(const char *path, const char *mode)
    Apre il file specificato da path.
FILE *fdopen(int fildes, const char *mode)
    Associa uno stream al file descriptor fildes.
FILE *freopen(const char *path, const char *mode, FILE *stream)
    Apre il file specificato da path associandolo allo stream specificato da stream, se questo è
    già aperto prima lo chiude.
```

Le funzioni ritornano un puntatore valido in caso di successo e `NULL` in caso di errore, in tal 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 nella modalità specificata da `mode`, che è una stringa che deve iniziare con almeno uno dei valori indicati in tab. 7.1 (sono possibili varie estensioni che vedremo in seguito).

L'uso più comune di `freopen` è per redirigere uno dei tre file standard (vedi sez. 7.1.3): il file `path` viene associato a `stream` e se questo è uno stream già aperto viene preventivamente chiuso.

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.

Valore	Significato
r	Il file viene aperto, l'accesso viene posto in sola lettura, lo stream è posizionato all'inizio del file.
r+	Il file viene aperto, l'accesso viene posto in lettura e scrittura, lo stream 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 stream 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 stream 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.
x	l'apertura fallisce se il file esiste già.

**Tabella 7.1:** Modalità di apertura di uno stream dello standard ANSI C che sono sempre presenti in qualunque sistema POSIX.

In realtà lo standard ANSI C prevede un totale di 15 possibili valori diversi per `mode`, ma in tab. 7.1 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.

Le *glibc* supportano alcune estensioni, queste devono essere sempre indicate dopo aver specificato il `mode` con uno dei valori di tab. 7.1. L'uso del carattere `x` serve per evitare di sovrascrivere

<sup>4</sup>`fopen` e `freopen` fanno parte dello standard ANSI C, `fdopen` è parte dello standard POSIX.1.

un file già esistente (è analoga all'uso dell'opzione `O_EXCL` in `open`), se il file specificato già esiste e si aggiunge questo carattere a `mode` la `fopen` fallisce.

Un'altra estensione serve a supportare la localizzazione, quando si aggiunge a `mode` una stringa della forma ",`ccs=STRING`" 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. 7.2.2) sono cancellate. Il file non viene duplicato e verrà chiuso alla chiusura dello stream.

I nuovi file saranno creati secondo quanto visto in sez. 5.3.4 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 di `umask` per il processo (si veda sez. 5.3.7).

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 rilettura; 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 aperto lo stream, si può cambiare la modalità di bufferizzazione (si veda sez. 7.3.2) fintanto che non si è effettuato alcuna operazione di I/O sul file.

Uno stream viene chiuso con la funzione `fclose` il cui prototipo è:

```
#include <stdio.h>
int fclose(FILE *stream)
    Chiude lo stream stream.
```

Restituisce 0 in caso di successo e `EOF` in caso di errore, nel qual caso imposta `errno` a `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 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 dalle *glibc*; se si vuole essere sicuri che il kernel forzi la scrittura su disco occorrerà effettuare una `sync` (vedi sez. 6.3.3).

Linux supporta anche una altra funzione, `fcloseall`, come estensione GNU implementata dalle *glibc*, accessibile avendo definito `_GNU_SOURCE`, il suo prototipo è:

```
#include <stdio.h>
int fcloseall(void)
    Chiude tutti gli stream.
```

Restituisce 0 se non ci sono errori ed `EOF` altrimenti.

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).

### 7.2.2 Lettura e scrittura 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 modalità di input/output non formattato:

1. *binario* in cui legge/scrive un blocco di dati alla volta, vedi sez. 7.2.3.
2. *a caratteri* in cui si legge/scrive un carattere alla volta (con la bufferizzazione gestita automaticamente dalla libreria), vedi sez. 7.2.4.
3. *di linea* in cui si legge/scrive una linea alla volta (terminata dal carattere di newline '\n'), vedi sez. 7.2.5.

ed inoltre la modalità di input/output formattato.

A differenza dell’interfaccia dei file descriptor, con gli stream il raggiungimento della fine del file è considerato un errore, e viene notificato come tale dai valori di uscita delle varie funzioni. Nella maggior parte dei casi questo avviene con la restituzione del valore intero (di tipo `int`) `EOF`<sup>5</sup> definito anch’esso nell’header `stdlib.h`.

Dato che le funzioni dell’interfaccia degli stream sono funzioni di libreria che si appoggiano a delle system call, esse non impostano direttamente la variabile `errno`, che mantiene il valore impostato dalla system call che ha riportato l’errore.

Siccome la condizione di end-of-file è anch’essa segnalata come errore, nasce il problema di come distinguerla da un errore effettivo; 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. 8.5.1 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:

```
#include <stdio.h>
int feof(FILE *stream)
    Controlla il flag di end-of-file di stream.
int ferror(FILE *stream)
    Controlla il flag di errore di stream.
```

Entrambe le funzioni ritornano un valore diverso da zero se i relativi flag sono impostati.

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 quindi deve 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:

```
#include <stdio.h>
void clearerr(FILE *stream)
    Cancella i flag di errore ed end-of-file di stream.
```

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` che non esegue il blocco dello stream (vedi sez. 7.3.3).

<sup>5</sup>la costante deve essere negativa, le *glibc* usano -1, altre implementazioni possono avere valori diversi.

### 7.2.3 Input/output binario

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 loro prototipi sono:

```
#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)
    Rispettivamente leggono e scrivono nmemb elementi di dimensione size dal buffer ptr al file stream.

Entrambe 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.
```

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 nelem)
{
    int size, nread;
    size = sizeof(*vec);
    if ((nread = fwrite(vec, size, nelem, stream)) != nelem) {
        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 \* nelem**; la sola differenza è che le funzioni non ritornano il numero di byte scritti, ma il numero di elementi.

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 (in genere 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.

Le *glibc* definiscono altre due funzioni per l'I/O binario, `fread_unlocked` e `fwrite_unlocked` che evitano il lock implicito dello stream, usato per dalla librerie per la gestione delle applicazioni multi-thread (si veda sez. 7.3.3 per i dettagli), i loro prototipi sono:

```
#include <stdio.h>
size_t fread_unlocked(void *ptr, size_t size, size_t nmemb, FILE *stream)
size_t fwrite_unlocked(const void *ptr, size_t size, size_t nmemb, FILE *stream)
```

Le funzioni sono identiche alle analoghe `fread` e `fwrite` ma non acquisiscono il lock implicito sullo stream.

entrambe le funzioni sono estensioni GNU previste solo dalle *glibc*.

#### 7.2.4 Input/output a caratteri

La seconda modalità di input/output 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`, i rispettivi prototipi sono:

```
#include <stdio.h>
int getc(FILE *stream)
    Legge un byte da stream e lo restituisce come intero. In genere è implementata come una
    macro.
int fgetc(FILE *stream)
    Legge un byte da stream e lo restituisce come intero. È sempre una funzione.
int getchar(void)
    Equivalente a getc(stdin).
```

Tutte queste funzioni leggono un byte alla volta, che viene restituito come intero; in caso di errore o fine del file il valore di ritorno è `EOF`.

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 parametro 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:

```
#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream)
    Legge un carattere esteso da stream. In genere è implementata come una macro.
wint_t fgetwc(FILE *stream)
    Legge un carattere esteso da stream. È una sempre una funzione.
wint_t getwchar(void)
    Equivalente a getwc(stdin).
```

Tutte queste funzioni leggono un carattere alla volta, in caso di errore o fine del file il valore di ritorno è `WEOF`.

Per scrivere un carattere si possono usare tre funzioni, analoghe alle precedenti usate per leggere: `putc`, `fputc` e `putchar`; i loro prototipi sono:

```
#include <stdio.h>
int putc(int c, FILE *stream)
    Scrive il carattere c su stream. In genere è implementata come una macro.
int fputc(FILE *stream)
    Scrive il carattere c su stream. È una sempre una funzione.
int putchar(void)
    Equivalente a putc(stdin).
```

Le funzioni scrivono sempre un carattere alla volta, il cui valore viene restituito in caso di successo; in caso di errore o fine del file il valore di ritorno è `EOF`.

Tutte queste funzioni scrivono sempre un byte alla volta, anche se prendono come parametro 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` le *glibc* provvedono 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:

```
#include <stdio.h>
int getw(FILE *stream)
    Legge una parola da stream.
int putw(int w, FILE *stream)
    Scrive la parola w su stream.
```

Le funzioni restituiscono la parola `w`, o `EOF` in caso di errore o di fine del file.

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 ridiventando 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)
    Rimanda indietro il carattere c, con un cast a unsigned char, sullo stream stream.
```

La funzione ritorna c in caso di successo e EOF in caso di errore.

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. 7.2.7) i caratteri rimandati indietro vengono scartati.

### 7.2.5 Input/output di linea

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 una modalità molto usata per l'I/O da terminale, ma 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)
    Scrive su string una linea letta da stdin.
char *fgets(char *string, int size, FILE *stream)
    Scrive su string la linea letta da stream per un massimo di size byte.
```

Le funzioni restituiscono l'indirizzo `string` in caso di successo o `NULL` in caso di errore.

Entrambe le funzioni effettuano la lettura (dal file specificato `fgets`, dallo standard input `gets`) di una linea di caratteri (terminata dal carattere *newline*, '\n', quello mappato sul tasto di ritorno a capo della tastiera), ma `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 `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.<sup>6</sup>

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 (in genere uno *shell code* cioè una sezione di programma che lancia una shell).

<sup>6</sup>questa tecnica è spiegata in dettaglio e con molta efficacia nell'ormai famoso articolo di Aleph1 [5].

La funzione `fgets` non ha i precedenti problemi di `gets` in quanto prende in input 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(const char *string)
    Scrive su stdout la linea string.
int fputs(const char *string, FILE *stream)
    Scrive su stream la linea string.
```

Le funzioni restituiscono un valore non negativo in caso di successo o `EOF` in caso di errore.

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)
    Legge un massimo di n caratteri estesi dal file stream al buffer ws.
int fputws(const wchar_t *ws, FILE *stream)
    Scrive la linea ws di caratteri estesi sul file stream.
```

Le funzioni ritornano rispettivamente `ws` o un numero non negativo in caso di successo e `NULL` o `EOF` in caso di errore o fine del file.

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.

Come per l'I/O binario e quello a caratteri, anche per l'I/O di linea le *glibc* supportano una serie di altre funzioni, estensioni di tutte quelle illustrate finora (eccetto `gets` e `puts`), che eseguono esattamente le stesse operazioni delle loro equivalenti, evitando però il lock implicito dello stream (vedi sez. 7.3.3). Come per le altre forme di I/O, dette funzioni hanno lo stesso nome della loro analoga normale, con l'aggiunta dell'estensione `_unlocked`.

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 dicasì quando si deve gestire il caso di stringa che eccede le dimensioni del buffer.

Per questo motivo le *glibc* prevedono, 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:

```
#include <stdio.h>
ssize_t getline(char **buffer, size_t *n, FILE *stream)
    Legge una linea dal file stream copiandola sul buffer indicato da buffer riallocandolo se necessario (l'indirizzo del buffer e la sua dimensione vengono sempre riscritte).
```

La funzione ritorna il numero di caratteri letti in caso di successo e -1 in caso di errore o di raggiungimento della fine del file.

La funzione permette di eseguire una lettura senza doversi preoccupare della eventuale lunghezza eccessiva della stringa da leggere. Essa prende come primo parametro l'indirizzo del puntatore al buffer su cui si vuole copiare la linea. Quest'ultimo *dove* essere stato allocato in precedenza con una `malloc` (non si può passare l'indirizzo di un puntatore ad una variabile locale); come secondo parametro la funzione vuole l'indirizzo della variabile contenente le dimensioni del buffer suddetto.

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 passata indietro (si noti infatti come per entrambi i parametri si siano usati dei *value result argument*, passando dei puntatori anziché i valori delle variabili, secondo la tecnica spiegata 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 `ptr` con una `free`.

Il valore di ritorno della funzione indica il numero di caratteri letti dallo stream (quindi compreso il newline, ma non lo zero di terminazione); questo permette anche di distinguere eventuali zeri letti dallo stream da quello inserito dalla funzione per terminare la linea. Se si è alla fine del file e non si è potuto leggere nulla o c'è stato un errore la funzione restituisce -1.

La seconda estensione GNU è una generalizzazione di `getline` per poter usare come separatore un carattere qualsiasi, la funzione si chiama `getdelim` ed il suo prototipo è:

```
#include <stdio.h>
ssize_t getdelim(char **buffer, size_t *n, int delim, FILE *stream)
    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 questa passando '\n' come valore di `delim`).

### 7.2.6 L'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, ...)
    Stampa su stdout gli argomenti, secondo il formato specificato da format.
int fprintf(FILE *stream, const char *format, ...)
    Stampa su stream gli argomenti, secondo il formato specificato da format.
int sprintf(char *str, const char *format, ...)
    Stampa sulla stringa str gli argomenti, secondo il formato specificato da format.
```

Le funzioni ritornano il numero di caratteri stampati.

le prime due servono per stampare su file (lo standard output o quello specificato) la terza permette di stampare 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, ...)
    Identica a sprintf, ma non scrive su str più di size caratteri.
```

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 dei parametri che dovranno essere passati a seguire (si noti come tutte queste funzioni siano *variadic*, prendendo un numero di argomenti variabile che dipende appunto da quello che si è specificato in `format`).

Valore	Tipo	Significato
%d	int	Stampa un numero intero in formato decimale con segno
%i	int	Identico a %i 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 appropriata 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 7.2:** Valori possibili per gli specificatori di conversione in una stringa di formato di `printf`.

La stringa è costituita da caratteri normali (tutti eccetto %), che vengono passati invariati all'output, 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. 7.2) che la conclude.

Il formato di una direttiva di conversione prevede una serie di possibili elementi opzionali oltre al % e allo specificatore di conversione. In generale essa è sempre del tipo:

% [n. parametro \$] [flag] [[larghezza] [. precisione]] [tipo] conversione

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 7.3:** I valori dei flag per il formato di `printf`

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 \$),
- uno o più flag (i cui valori possibili sono riassunti in tab. 7.3) 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. 7.4).

Dettagli ulteriori sulle varie opzioni possono essere trovati nella pagina di manuale di `printf` e nella documentazione delle *glibc*.

Valore	Significato
hh	una conversione intera corrisponde a un <code>char</code> con o senza segno, o il puntatore per il numero dei parametri n è di tipo <code>char</code> .
h	una conversione intera corrisponde a uno <code>short</code> con o senza segno, o il puntatore per il numero dei parametri n è di tipo <code>short</code> .
l	una conversione intera corrisponde a un <code>long</code> con o senza segno, o il puntatore per il numero dei parametri n è di tipo <code>long</code> , o il carattere o la stringa seguenti sono in formato esteso.
ll	una conversione intera corrisponde a un <code>long long</code> con o senza segno, o il puntatore per il numero dei parametri n è di tipo <code>long long</code> .
L	una conversione in virgola mobile corrisponde a un <code>double</code> .
q	sinonimo di ll.
j	una conversione intera corrisponde a un <code>intmax_t</code> o <code>uintmax_t</code> .
z	una conversione intera corrisponde a un <code>size_t</code> o <code>ssize_t</code> .
t	una conversione intera corrisponde a un <code>ptrdiff_t</code> .

**Tabella 7.4:** 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 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)
    Stampa su stdout gli argomenti della lista ap, secondo il formato specificato da format.
int vfprintf(FILE *stream, const char *format, va_list ap)
    Stampa su stream gli argomenti della lista ap, secondo il formato specificato da format.
int vsprintf(char *str, const char *format, va_list ap)
    Stampa sulla stringa str gli argomenti della lista ap, secondo il formato specificato da format.
```

Le funzioni ritornano il numero di caratteri stampati.

con queste funzioni diventa possibile selezionare gli argomenti che si vogliono passare ad una routine di stampa, passando direttamente la lista tramite il parametro ap. Per poter far questo

ovviamente la lista dei parametri dovrà essere opportunamente trattata (l'argomento è esaminato in sez. 2.4.2), e dopo l'esecuzione della funzione `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 `vsnprintf` esiste una analoga `vsnprintf` che pone un limite sul numero di caratteri che vengono scritti sulla stringa di destinazione:

```
#include <stdio.h>
vsnprintf(char *str, size_t size, const char *format, va_list ap)
    Identica a vsprintf, ma non scrive su str più di size caratteri.
```

in modo da evitare possibili buffer overflow.

Per eliminare alla radice questi problemi, le *glibc* supportano 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, ...)
    Stampa gli argomenti specificati secondo il formato specificato da format su una stringa allocata automaticamente all'indirizzo *strptr.
int vasprintf(char **strptr, const char *format, va_list ap)
    Stampa gli argomenti della lista ap secondo il formato specificato da format su una stringa allocata automaticamente all'indirizzo *strptr.
```

Le funzioni ritornano il numero di caratteri stampati.

Entrambe le funzioni prendono come parametro `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 delle *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 una scansione di stdin cercando una corrispondenza di quanto letto con il formato dei dati specificato da format, ed effettua le relative conversione memorizzando il risultato nei parametri seguenti.
int fscanf(FILE *stream, const char *format, ...)
    Analoga alla precedente, ma effettua la scansione su stream.
int sscanf(char *str, const char *format, ...)
    Analoga alle precedenti, ma effettua la scansione dalla stringa str.
```

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. In caso di errore o fine del file viene invece restituito `EOF`.

e come per le analoghe funzioni di scrittura esistono le relative `vscanf`, `vfscanf` `vsscanf` che usano un puntatore ad una lista di argomenti.

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 parametro 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 etc.), 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 delle *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 corrispondenze, 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`<sup>7</sup> per generare un analizzatore lessicale o il `bison`<sup>8</sup> per generare un parser.

### 7.2.7 Posizionamento su uno stream

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, quando cioè si ha a che fare con quello che viene detto un file ad *accesso casuale*.<sup>9</sup>

In GNU/Linux ed in generale in ogni sistema unix-like la posizione nel file 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 in VMS può essere rappresentata come 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 parametri di tipo diverso. Le funzioni tradizionali usate per il riposizionamento della posizione in uno stream sono `fseek` e `rewind` i cui prototipi sono:

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int whence)
    Sposta la posizione nello stream secondo quanto specificato tramite offset e whence.
void rewind(FILE *stream)
    Riporta la posizione nello stream all'inizio del file.
```

L’uso di `fseek` è del tutto analogo a quello di `lseek` per i file descriptor, ed i parametri, a parte il tipo, hanno lo stesso significato; in particolare `whence` assume gli stessi valori già visti in sez. 6.2.3. La funzione restituisce 0 in caso di successo e -1 in caso di errore. La funzione `rewind` riporta semplicemente la posizione corrente all’inizio dello stream, ma non esattamente

<sup>7</sup>il programma `flex`, è una implementazione libera di `lex` un generatore di analizzatori lessicali. Per i dettagli si può fare riferimento al manuale [6].

<sup>8</sup>il programma `bison` è un clone del generatore di parser `yacc`, maggiori dettagli possono essere trovati nel relativo manuale [7].

<sup>9</sup>dato che in un sistema Unix esistono vari tipi di file, come le fifo ed i file di dispositivo, non è scontato che questo sia sempre vero.

equivalente ad una `fseek(stream, 0L, SEEK_SET)` in quanto vengono cancellati anche i flag di errore e fine del file.

Per ottenere la posizione corrente si usa invece la funzione `ftell`, il cui prototipo è:

```
#include <stdio.h>
long ftell(FILE *stream)
    Legge la posizione attuale nello stream stream.
```

La funzione restituisce la posizione corrente, o -1 in caso di fallimento, che può esser dovuto sia al fatto che il file non supporta il riposizionamento che al fatto che la posizione non può essere espressa con un `long int`

la funzione restituisce la posizione come numero di byte dall'inizio dello stream.

Queste funzioni esprimono tutte la posizione nel file come un `long int`. Dato che (ad esempio quando si usa un filesystem indicizzato a 64 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:

```
#include <stdio.h>
int fsetpos(FILE *stream, fpos_t *pos)
    Imposta la posizione corrente nello stream stream al valore specificato da pos.
int fgetpos(FILE *stream, fpos_t *pos)
    Legge la posizione corrente nello stream stream e la scrive in pos.
```

Le funzioni ritornano 0 in caso di successo e -1 in caso di errore.

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`.

## 7.3 Funzioni avanzate

In questa sezione esamineremo alcune funzioni avanzate che permettono di eseguire operazioni particolari sugli stream, come leggerne gli attributi, controllarne le modalità di bufferizzazione, gestire direttamente i lock impliciti per la programmazione multi thread.

### 7.3.1 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 quest'ultimo, il prototipo della funzione è:

```
#include <stdio.h>
int fileno(FILE *stream)
    Legge il file descriptor sottostante lo stream stream.
```

Restituisce il numero del file descriptor in caso di successo, e -1 qualora `stream` non sia valido, nel qual caso imposta `errno` a `EBADF`.

ed in questo modo diventa possibile usare direttamente `fcntl`.

Questo permette di accedere agli attributi del file descriptor sottostante lo stream, ma non ci dà nessuna informazione riguardo alle proprietà dello stream medesimo. Le *glibc* però supportano alcune estensioni derivate da Solaris, che permettono di ottenere informazioni utili.

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 si deve ricordare come il file è stato aperto. 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 può essere evitato con le due funzioni `__freadable` e `__fwriteable` i cui prototipi sono:

```
#include <stdio_ext.h>
int __freadable(FILE *stream)
    Restituisce un valore diverso da zero se stream consente la lettura.
int __fwriteable(FILE *stream)
    Restituisce un valore diverso da zero se stream consente la scrittura.
```

che permettono di ottenere questa informazione.

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 `__fwriteing` servono a tale scopo, il loro prototipo è:

```
#include <stdio_ext.h>
int __freading(FILE *stream)
    Restituisce un valore diverso da zero se stream è aperto in sola lettura o se l'ultima
    operazione è stata di lettura.
int __fwriteing(FILE *stream)
    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.

### 7.3.2 Il controllo della bufferizzazione

Come accennato in sez. 7.1.4 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 suo prototipo è:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
    Imposta la bufferizzazione dello stream stream nella modalità indicata da mode, usando
    buf come buffer di lunghezza size.
    Restituisce zero in caso di successo, ed un valore qualunque in caso di errore, nel qual caso errno
    viene impostata opportunamente.
```

La funzione 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**.

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 disallocarlo 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 macro `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 disallocazione 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.

Valore	Modalità
<code>_IONBF</code>	<i>unbuffered</i>
<code>_IOLBF</code>	<i>line buffered</i>
<code>_IOFBF</code>	<i>fully buffered</i>

**Tabella 7.5:** Valori del parametro `mode` di `setvbuf` per l'impostazione delle modalità di bufferizzazione.

Per evitare che `setvbuf` imposti il buffer basta passare un valore `NULL` per `buf` e la funzione ignorerà il parametro `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. 7.5. Qualora si specifichi la modalità non bufferizzata i valori di `buf` e `size` vengono sempre ignorati.

Oltre a `setvbuf` le *glibc* definiscono altre tre funzioni per la gestione della bufferizzazione di uno stream: `setbuf`, `setbuffer` e `setlinebuf`; i loro prototipi sono:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf)
    Disabilita la bufferizzazione se buf è NULL, altrimenti usa buf come buffer di dimensione
    BUFSIZ in modalità fully buffered.
void setbuffer(FILE *stream, char *buf, size_t size)
    Disabilita la bufferizzazione se buf è NULL, altrimenti usa buf come buffer di dimensione
    size in modalità fully buffered.
void setlinebuf(FILE *stream)
    Pone lo stream in modalità line buffered.
```

tutte queste funzioni sono realizzate con opportune chiamate a `setvbuf` e sono definite solo per compatibilità con le vecchie librerie BSD. Infine le *glibc* provvedono le funzioni non standard<sup>10</sup> `__flbf` e `__fbuflen` che permettono di leggere le proprietà di bufferizzazione di uno stream; i cui prototipi sono:

```
#include <stdio_ext.h>
int __flbf(FILE *stream)
    Restituisce un valore diverso da zero se stream è in modalità line buffered.
size_t __fbuflen(FILE *stream)
    Restituisce le dimensioni del buffer di stream.
```

Come già accennato, indipendentemente dalla modalità di bufferizzazione scelta, si può forzare lo scarico dei dati sul file con la funzione `fflush`, il suo prototipo è:

```
#include <stdio.h>
int fflush(FILE *stream)
    Forza la scrittura di tutti i dati bufferizzati dello stream stream.

Restituisce zero in caso di successo, ed EOF in caso di errore, impostando errno a EBADF se stream
non è aperto o non è aperto in scrittura, o ad uno degli errori di write.
```

anche di questa funzione esiste una analoga `fflush_unlocked`<sup>11</sup> che non effettua il blocco dello stream.

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

<sup>10</sup>anche queste funzioni sono originarie di Solaris.

<sup>11</sup>accessibile definendo `_BSD_SOURCE` o `_SVID_SOURCE` o `_GNU_SOURCE`.

buffered; per questo motivo le *glibc* supportano una estensione di Solaris, la funzione `_flushlbf`, il cui prototipo è:

```
#include <stdio-ext.h>
void _flushlbf(void)
    Forza la scrittura di tutti i dati bufferizzati degli stream in modalità line buffered.
```

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. 6.3.3).

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 input e di output dello stream stream.
    Restituisce zero in caso di successo, ed EOF in caso di 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`.

### 7.3.3 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 delle funzioni `flockfile`, `ftrylockfile` 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)
    Esegue l'acquisizione del lock dello stream stream, bloccandosi se il lock non è disponibile.
int ftrylockfile(FILE *stream)
    Tenta l'acquisizione del lock dello stream stream, senza bloccarsi se il lock non è disponibile.
    Ritorna zero in caso di acquisizione del lock, diverso da zero altrimenti.
void funlockfile(FILE *stream)
    Rilascia il lock dello stream stream.
```

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 abbiamo visto come alle usuali funzioni di I/O non formattato siano associate delle versioni `_unlocked` (alcune previste dallo stesso standard POSIX, altre

aggiunte come estensioni dalle *glibc*) che possono essere usate quando il locking non serve<sup>12</sup> 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 le *glibc* forniscono al programmatore pigro un'altra via<sup>13</sup> 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 o richiede a seconda del valore di type la modalità in cui le operazioni di I/O su
    stream vengono effettuate rispetto all'acquisizione implicita del blocco sullo stream.

Restituisce lo stato di locking interno dello stream con uno dei valori FSETLOCKING_INTERNAL o
FSETLOCKING_BYCALLER.
```

La funzione imposta o legge lo stato della modalità di operazione di uno stream nei confronti del locking a seconda del valore specificato con `type`, che può essere uno dei seguenti:

`FSETLOCKING_INTERNAL` Lo stream 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 stream.

`FSETLOCKING_QUERY` Restituisce lo stato corrente della modalità di blocco dello stream.

---

<sup>12</sup>in 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.

<sup>13</sup>anche questa mutuata da estensioni introdotte in Solaris.



# Capitolo 8

## 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.

### 8.1 Capacità e caratteristiche del sistema

In questa sezione tratteremo le varie modalità con cui un programma può ottenere informazioni riguardo alle capacità del sistema. 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.

La definizione di queste caratteristiche ed il tentativo di provvedere 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, spesso 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.

#### 8.1.1 Limiti e parametri di 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 header file 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 in sez. 8.1.2).

Lo standard ANSI C definisce dei limiti che sono tutti fissi, pertanto questi saranno sempre disponibili al momento della compilazione. Un elenco, ripreso da `limits.h`, è riportato in tab. 8.1. 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`). 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.

Costante	Valore	Significato
<code>MB_LEN_MAX</code>	16	massima dimensione di un carattere esteso
<code>CHAR_BIT</code>	8	bit di <code>char</code>
<code>UCHAR_MAX</code>	255	massimo di <code>unsigned char</code>
<code>SCHAR_MIN</code>	-128	minimo di <code>signed char</code>
<code>SCHAR_MAX</code>	127	massimo di <code>signed char</code>
<code>CHAR_MIN</code>	<sup>1</sup>	minimo di <code>char</code>
<code>CHAR_MAX</code>	<sup>2</sup>	massimo di <code>char</code>
<code>SHRT_MIN</code>	-32768	minimo di <code>short</code>
<code>SHRT_MAX</code>	32767	massimo di <code>short</code>
<code>USHRT_MAX</code>	65535	massimo di <code>unsigned short</code>
<code>INT_MAX</code>	2147483647	minimo di <code>int</code>
<code>INT_MIN</code>	-2147483648	minimo di <code>int</code>
<code>UINT_MAX</code>	4294967295	massimo di <code>unsigned int</code>
<code>LONG_MAX</code>	2147483647	massimo di <code>long</code>
<code>LONG_MIN</code>	-2147483648	minimo di <code>long</code>
<code>ULONG_MAX</code>	4294967295	massimo di <code>unsigned long</code>

**Tabella 8.1:** Costanti definite in `limits.h` in conformità allo standard ANSI C.

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. 8.2.

Costante	Valore	Significato
<code>LLONG_MAX</code>	9223372036854775807	massimo di <code>long long</code>
<code>LLONG_MIN</code>	-9223372036854775808	minimo di <code>long long</code>
<code>ULLONG_MAX</code>	18446744073709551615	massimo di <code>unsigned long long</code>

**Tabella 8.2:** Macro definite in `limits.h` in conformità allo standard ISO C90.

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. 8.1.3.

Purtroppo la sezione dello standard che tratta questi argomenti è una delle meno chiare<sup>3</sup>. Lo standard prevede che ci siano 13 macro che descrivono le caratteristiche del sistema (7 per le caratteristiche generiche, riportate in tab. 8.3, e 6 per le caratteristiche dei file, riportate in tab. 8.7).

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

<sup>1</sup>il valore può essere 0 o `SCHAR_MIN` a seconda che il sistema usi caratteri con segno o meno.

<sup>2</sup>il valore può essere `UCHAR_MAX` o `SCHAR_MAX` a seconda che il sistema usi caratteri con segno o meno.

<sup>3</sup>tanto che Stevens, in [1], la porta come esempio di “standardese”.

Costante	Valore	Significato
ARG_MAX	131072	dimensione massima degli argomenti passati ad una funzione della famiglia <code>exec</code> .
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 <code>timezone</code> (vedi sez. 8.4.3)).
NGROUPS_MAX	32	numero di gruppi supplementari per processo (vedi sez. 3.3.1).
SSIZE_MAX	32767	valore massimo del tipo <code>ssize_t</code> .

**Tabella 8.3:** Costanti per i limiti del sistema.

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; detti valori sono riportati in tab. 8.4.

Costante	Valore	Significato
_POSIX_ARG_MAX	4096	dimensione massima degli argomenti passati ad una funzione della famiglia <code>exec</code> .
_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		dimensione massima del nome di una <code>timezone</code> (vedi sez. 8.4.4).
_POSIX_NGROUPS_MAX	0	numero di gruppi supplementari per processo (vedi sez. 3.3.1).
_POSIX_SSIZET_MAX	32767	valore massimo del tipo <code>ssize_t</code> .
_POSIX_AIO_LISTIO_MAX	2	
_POSIX_AIO_MAX	1	

**Tabella 8.4:** Macro dei valori minimi delle caratteristiche generali del sistema per la conformità allo standard POSIX.1.

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 oggigiorno. Per questo è sempre meglio utilizzare i valori ottenuti da `sysconf`.

Macro	Significato
_POSIX_JOB_CONTROL	il sistema supporta il <i>job control</i> (vedi sez. 10.1).
_POSIX_SAVED_IDS	il sistema supporta gli identificatori del gruppo <i>saved</i> (vedi sez. 3.3.1) per il controllo di accesso dei processi
_POSIX_VERSION	fornisce la versione dello standard POSIX.1 supportata nel formato YYYYMM (ad esempio 199009L).

**Tabella 8.5:** Alcune macro definite in `limits.h` in conformità allo standard POSIX.1.

Oltre ai precedenti valori (e a quelli relativi ai file elencati in tab. 8.8), che devono essere obbligatoriamente definiti, lo standard POSIX.1 ne prevede parecchi 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 menzionare alcune macro di uso comune, (riportate in tab. 8.5), 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*).

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 delle *glibc*.

### 8.1.2 La funzione sysconf

Come accennato in sez. 8.1.1 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 a run time, è necessario ottenerne il valore attraverso la funzione `sysconf`. Il prototipo di questa funzione è:

```
#include <unistd.h>
long sysconf(int name)
    Restituisce il valore del parametro di sistema name.
```

La funzione restituisce indietro il valore del parametro richiesto, o 1 se si tratta di un'opzione disponibile, 0 se l'opzione non è disponibile e -1 in caso di errore (ma `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. 8.6; l'elenco completo è contenuto in `bits/confname.h`, ed una lista più esaustiva, con le relative spiegazioni, si può trovare nel manuale delle *glibc*.

Parametro	Macro sostituita	Significato
<code>_SC_ARG_MAX</code>	<code>ARG_MAX</code>	La dimensione massima degli argomenti passati ad una funzione della famiglia <code>exec</code> .
<code>_SC_CHILD_MAX</code>	<code>_CHILD_MAX</code>	Il numero massimo di processi contemporanei che un utente può eseguire.
<code>_SC_OPEN_MAX</code>	<code>_OPEN_MAX</code>	Il numero massimo di file che un processo può mantenere aperti in contemporanea.
<code>_SC_STREAM_MAX</code>	<code>STREAM_MAX</code>	Il massimo numero di stream che un processo può mantenere aperti in contemporanea. Questo limite previsto anche dallo standard ANSI C, che specifica la macro <code>FOPEN_MAX</code> .
<code>_SC_TZNAME_MAX</code>	<code>TZNAME_MAX</code>	La dimensione massima di un nome di una <code>timezone</code> (vedi sez. 8.4.4).
<code>_SC_NGROUPS_MAX</code>	<code>NGROUP_MAX</code>	Massimo numero di gruppi supplementari che può avere un processo (vedi sez. 3.3.1).
<code>_SC_SSIZET_MAX</code>	<code>SSIZE_MAX</code>	valore massimo del tipo di dato <code>ssize_t</code> .
<code>_SC_CLK_TCK</code>	<code>CLK_TCK</code>	Il numero di <i>clock tick</i> al secondo, cioè l'unità di misura del <i>process time</i> (vedi sez. 8.4.1).
<code>_SC_JOB_CONTROL</code>	<code>_POSIX_JOB_CONTROL</code>	Indica se è supportato il <i>job control</i> (vedi sez. 10.1) in stile POSIX.
<code>_SC_SAVED_IDS</code>	<code>_POSIX_SAVED_IDS</code>	Indica se il sistema supporta i <i>saved id</i> (vedi sez. 3.3.1).
<code>_SC_VERSION</code>	<code>_POSIX_VERSION</code>	Indica il mese e l'anno di approvazione della revisione dello standard POSIX.1 a cui il sistema fa riferimento, nel formato YYYYMM, la revisione più recente è 199009L, che indica il Settembre 1990.

**Tabella 8.6:** Parametri del sistema leggibili dalla funzione `sysconf`.

In generale ogni limite o caratteristica del sistema per cui è definita una macro, sia dagli standard ANSI C e ISO C90, che da POSIX.1 e POSIX.2, può essere ottenuto attraverso una chiamata a `sysconf`. Il valore si otterrà specificando come valore del parametro `name` il nome ottenuto aggiungendo `_SC_` ai nomi delle macro definite dai primi due, o sostituendolo a `_POSIX_` per le macro definite dagli altri due.

In generale si dovrebbe fare uso di `sysconf` solo quando la relativa macro non è definita, quindi con un codice analogo al seguente:

```
get_child_max(void)
{
#ifndef CHILD_MAX
    return CHILD_MAX;
#else
    int val = sysconf(_SC_CHILD_MAX);
    if (val < 0) {
        perror("fatal error");
        exit(-1);
    }
    return val;
}
```

ma in realtà in Linux queste macro sono comunque definite, indicando però un limite generico. Per questo motivo è sempre meglio usare i valori restituiti da `sysconf`.

### 8.1.3 I limiti 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. 8.7.

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 pipe (vedi sez. 12.1.1).
MAX_CANON	255	dimensione di una riga di terminale in modo canonico (vedi sez. 10.2.1).
MAX_INPUT	255	spazio disponibile nella coda di input del terminale (vedi sez. 10.2.1).

*Tabella 8.7:* 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. 8.8, e per esse vale lo stesso discorso fatto per le analoghe di tab. 8.4.

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 pipe.
_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 8.8:* Costanti dei valori minimi delle caratteristiche dei file per la conformità allo standard POSIX.1.

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.

#### 8.1.4 La funzione pathconf

In generale i limiti per i file sono molto più soggetti ad essere variabili rispetto ai limiti generali del sistema; ad esempio parametri come la lunghezza del nome del file o il numero di link possono variare da filesystem a filesystem; per questo motivo questi limiti devono essere sempre controllati con la funzione `pathconf`, il cui prototipo è:

```
#include <unistd.h>
long pathconf(char *path, int name)
    Restituisce il valore del parametro name per il file path.
```

La funzione restituisce indietro il valore del parametro richiesto, o -1 in caso di errore (ed `errno` viene impostata ad uno degli errori possibili relativi all'accesso a `path`).

E si noti come la funzione in questo caso richieda un parametro che specifichi a quale file si fa riferimento, dato che il valore del limite cercato può variare a seconda del filesystem. 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 del parametro name per il file fd.
```

È identica a `pathconf` solo che utilizza un file descriptor invece di un *pathname*; pertanto gli errori restituiti cambiano di conseguenza.

ed il suo comportamento è identico a quello di `pathconf`.

#### 8.1.5 La funzione uname

Un'altra funzione che si può utilizzare per raccogliere informazioni sia riguardo al sistema che al computer su cui esso sta girando è `uname`; il suo prototipo è:

```
#include <sys/utsname.h>
int uname(struct utsname *info)
    Restituisce informazioni sul sistema nella struttura info.
```

La funzione ritorna 0 in caso di successo e -1 in caso di fallimento, nel qual caso `errno` assumerà il valore `EFAULT`.

La funzione, che viene usata dal comando `uname`, restituisce le informazioni richieste nella struttura `info`; anche questa struttura è definita in `sys/utsname.h`, secondo quanto mostrato in fig. 8.1, e le informazioni memorizzate nei suoi membri indicano rispettivamente:

- il nome del sistema operativo;
- il nome della release del kernel;
- il nome della versione del kernel;
- il tipo di macchina in uso;
- il nome della stazione;
- il nome del domino.

L'ultima informazione è stata aggiunta di recente e non è prevista dallo standard POSIX, essa è accessibile, come mostrato in fig. 8.1, solo definendo `_GNU_SOURCE`.

In generale si tenga presente che le dimensioni delle stringhe di una `utsname` non è specificata, e che esse sono sempre terminate con NUL; il manuale delle *glibc* indica due diverse dimensioni, `_UTSNAME_LENGTH` per i campi standard e `_UTSNAME_DOMAIN_LENGTH` per quello specifico per il

---

```

struct utsname {
    char sysname [];
    char nodename [];
    char release [];
    char version [];
    char machine [];
#ifdef _GNU_SOURCE
    char domainname [];
#endif
};

```

---

*Figura 8.1:* La struttura utsname.

nome di dominio; altri sistemi usano nomi diversi come SYS\_NMLN o \_SYS\_NMLN o UTSLEN che possono avere valori diversi.<sup>4</sup>

## 8.2 Opzioni e configurazione del sistema

Come abbiamo accennato nella sezione precedente, non tutti i limiti che caratterizzano il sistema sono fissi, o perlomeno non lo sono in tutte le implementazioni. Finora abbiamo visto come si può fare per leggerli, ci manca di esaminare il meccanismo che permette, quando questi possono variare durante l'esecuzione del sistema, di modificarli.

Inoltre, al di là di quelli che possono essere limiti caratteristici previsti da uno standard, ogni sistema può avere una sua serie di altri parametri di configurazione, che, non essendo mai fissi e variando da sistema a sistema, non sono stati inclusi nella standardizzazione della sezione precedente. Per questi occorre, oltre al meccanismo di impostazione, pure un meccanismo di lettura. Affronteremo questi argomenti in questa sezione, insieme alle funzioni che si usano per il controllo di altre caratteristiche generali del sistema, come quelle per la gestione dei filesystem e di utenti e gruppi.

### 8.2.1 La funzione sysctl ed il filesystem /proc

La funzione che permette la lettura ed l'impostazione dei parametri del sistema è **sysctl**; è una funzione derivata da BSD4.4, ma l'implementazione è specifica di Linux; il suo prototipo è:

```

#include <unistd.h>
int sysctl(int *name, int nlen, void *oldval, size_t *oldlenp, void *newval,
           size_t newlen)
Legge o scrive uno dei parametri di sistema.

```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà uno dei valori:

<b>EPERM</b>	non si ha il permesso di accedere ad uno dei componenti nel cammino specificato per il parametro, o di accedere al parametro nella modalità scelta.
<b>ENOTDIR</b>	non esiste un parametro corrispondente al nome <b>name</b> .
<b>EINVAL</b>	o si è specificato un valore non valido per il parametro che si vuole impostare o lo spazio provvisto per il ritorno di un valore non è delle giuste dimensioni.
<b>ENOMEM</b>	talvolta viene usato più correttamente questo errore quando non si è specificato sufficiente spazio per ricevere il valore di un parametro.
ed inoltre <b>EFAULT</b> .	

<sup>4</sup>Nel caso di Linux **uname** corrisponde in realtà a 3 system call diverse, le prime due usano rispettivamente delle lunghezze delle stringhe 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.

I parametri a cui la funzione permettere di accedere sono organizzati in maniera gerarchica all'interno di un albero;<sup>5</sup> per accedere ad uno di essi occorre specificare un cammino attraverso i vari nodi dell'albero, in maniera analoga a come avviene per la risoluzione di un *pathname* (da cui l'uso alternativo del filesystem `/proc`, che vedremo dopo).

Ciascun nodo dell'albero è identificato da un valore intero, ed il cammino che arriva ad identificare un parametro specifico è passato alla funzione attraverso l'array `name`, di lunghezza `nlen`, che contiene la sequenza dei vari nodi da attraversare. Ogni parametro ha un valore in un formato specifico che può essere un intero, una stringa o anche una struttura complessa, per questo motivo i valori vengono passati come puntatori `void`.

L'indirizzo a cui il valore corrente del parametro deve essere letto è specificato da `oldvalue`, e lo spazio ivi disponibile è specificato da `oldlenp` (passato come puntatore per avere indietro la dimensione effettiva di quanto letto); il valore che si vuole impostare nel sistema è passato in `newval` e la sua dimensione in `newlen`.

Si può effettuare anche una lettura e scrittura simultanea, nel qual caso il valore letto restituito dalla funzione è quello precedente alla scrittura.

I parametri accessibili attraverso questa funzione sono moltissimi, e possono essere trovati in `sysctl.h`, essi inoltre dipendono anche dallo stato corrente del kernel (ad esempio dai moduli che sono stati caricati nel sistema) e in genere i loro nomi possono variare da una versione di kernel all'altra; per questo è sempre il caso di evitare l'uso di `sysctl` quando esistono modalità alternative per ottenere le stesse informazioni. Alcuni esempi di parametri ottenibili 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

Come accennato in Linux si ha una modalità alternativa per accedere alle stesse informazioni di `sysctl` attraverso l'uso del filesystem `/proc`. Questo è un filesystem virtuale, generato direttamente dal kernel, che non fa riferimento a nessun dispositivo fisico, ma presenta in forma di file alcune delle strutture interne del kernel stesso.

In particolare l'albero dei valori di `sysctl` viene presentato in forma di file nella directory `/proc/sys`, cosicché è possibile accedervi specificando un *pathname* e leggendo e scrivendo sul file corrispondente al parametro scelto. Il kernel si occupa di generare al volo il contenuto ed i nomi dei file corrispondenti, e questo ha il grande vantaggio di rendere accessibili i vari parametri a qualunque comando di shell e di permettere la navigazione dell'albero dei valori.

Alcune delle corrispondenze dei file presenti in `/proc/sys` con i valori di `sysctl` sono riportate nei commenti del codice che può essere trovato in `linux/sysctl.h`,<sup>6</sup> la informazione disponibile in `/proc/sys` è riportata inoltre nella documentazione inclusa nei sorgenti del kernel, nella directory `Documentation/sysctl`.

Ma oltre alle informazioni ottenibili da `sysctl` dentro `proc` sono disponibili moltissime altre informazioni, fra cui ad esempio anche quelle fornite da `uname` (vedi sez. 8.2) che sono mantenute nei file `ostype`, `hostname`, `osrelease`, `version` e `domainname` di `/proc/kernel/`.

---

<sup>5</sup>si tenga presente che includendo solo `unistd.h`, saranno definiti solo i parametri generici; dato che ce ne sono molti specifici dell'implementazione, nel caso di Linux occorrerà includere anche i file `linux/unistd.h` e `linux/sysctl.h`.

<sup>6</sup>indicando un file di definizioni si fa riferimento alla directory standard dei file di include, che in ogni distribuzione che si rispetti è `/usr/include`.

### 8.2.2 La gestione delle proprietà dei filesystem

Come accennato in sez. 4.1.1 per poter accedere ai file occorre prima rendere disponibile al sistema il filesystem su cui essi sono memorizzati; l'operazione di attivazione del filesystem è chiamata *montaggio*, per far questo in Linux<sup>7</sup> si usa la funzione `mount` il cui prototipo è:

```
#include <sys/mount.h>
mount(const char *source, const char *target, const char *filesystemtype,
       unsigned long mountflags, const void *data)
    Monta il filesystem di tipo filesystemtype contenuto in source sulla directory target.
```

La funzione ritorna 0 in caso di successo e -1 in caso di fallimento, nel qual caso gli errori comuni a tutti i filesystem che possono essere restituiti in `errno` sono:

<code>EPERM</code>	il processo non ha i privilegi di amministratore.
<code>ENODEV</code>	<code>filesystemtype</code> non esiste o non è configurato nel kernel.
<code>ENOTBLK</code>	non si è usato un <i>block device</i> per <code>source</code> quando era richiesto.
<code>EBUSY</code>	<code>source</code> è già montato, o non può essere rimontato in read-only perché ci sono ancora file aperti in scrittura, o <code>target</code> è ancora in uso.
<code>EINVAL</code>	il device <code>source</code> presenta un <i>superblock</i> non valido, o si è cercato di rimontare un filesystem non ancora montato, o di montarlo senza che <code>target</code> sia un <i>mount point</i> o di spostarlo quando <code>target</code> non è un <i>mount point</i> o è <code>/</code> .
<code>EACCES</code>	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 averlo specificato o il device <code>source</code> è su un filesystem montato con l'opzione <code>MS_NODEV</code> .
<code>ENXIO</code>	il <i>major number</i> del device <code>source</code> è sbagliato.
<code>EMFILE</code>	la tabella dei device <code>dummy</code> è piena.

ed inoltre `ENOTDIR`, `EFAULT`, `ENOMEM`, `ENAMETOOLONG`, `ENOENT` o `ELOOP`.

La funzione monta sulla directory `target`, detta *mount point*, il filesystem contenuto in `source`. In generale un filesystem è contenuto su un disco, e l'operazione di montaggio corrisponde a rendere visibile al sistema il contenuto del suddetto disco, identificato attraverso il file di dispositivo ad esso associato.

Ma la struttura del virtual filesystem vista in sez. 4.2.1 è molto più 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 un filesystem, inoltre alcuni filesystem, come `proc` o `devfs` sono del tutto virtuali, i loro dati sono generati al volo ad ogni lettura, e passati al kernel ad ogni scrittura.

Il tipo di filesystem è specificato da `filesystemtype`, che deve essere una delle stringhe riportate nel file `/proc/filesystems`, che contiene l'elenco dei filesystem supportati dal kernel; nel caso si sia indicato uno dei filesystem virtuali, il contenuto di `source` viene ignorato.

Dopo l'esecuzione della funzione il contenuto del filesystem viene resto disponibile nella directory specificata come *mount point*, il precedente contenuto di detta directory viene mascherato dal contenuto della directory radice del filesystem montato.

Dal kernel 2.4.x inoltre è divenuto possibile sia spostare atomicamente un *mount point* da una directory ad un'altra, sia montare in diversi *mount point* lo stesso filesystem, sia montare più filesystem sullo stesso *mount point* (nel qual caso vale quanto appena detto, e solo il contenuto dell'ultimo filesystem montato sarà visibile).

Ciascun filesystem è dotato di caratteristiche specifiche che possono essere attivate o meno, alcune di queste sono generali (anche se non è detto siano disponibili in ogni filesystem), e vengono specificate come opzioni di montaggio con l'argomento `mountflags`.

In Linux `mountflags` deve essere un intero a 32 bit i cui 16 più significativi sono un *magic*

<sup>7</sup>la funzione è specifica di Linux e non è portabile.

*number*<sup>8</sup> mentre i 16 meno significativi sono usati per specificare le opzioni; essi sono usati come maschera binaria e vanno impostati con un OR aritmetico della costante `MS_MGC_VAL` con i valori riportati in tab. 8.9.

Parametro	Valore	Significato
<code>MS_RDONLY</code>	1	monta in sola lettura
<code>MS_NOSUID</code>	2	ignora i bit <i>suid</i> e <i>sgid</i>
<code>MS_NODEV</code>	4	impedisce l'accesso ai file di dispositivo
<code>MS_NOEXEC</code>	8	impedisce di eseguire programmi
<code>MS_SYNCHRONOUS</code>	16	abilita la scrittura sincrona
<code>MS_REMOUNT</code>	32	rimonta il filesystem cambiando i flag
<code>MS_MANDLOCK</code>	64	consente il <i>mandatory locking</i> (vedi sez. 11.4.5)
<code>S_WRITE</code>	128	scrive normalmente
<code>S_APPEND</code>	256	consente la scrittura solo in <i>append mode</i> (vedi sez. 6.3.1)
<code>S_IMMUTABLE</code>	512	impedisce che si possano modificare i file
<code>MS_NOATIME</code>	1024	non aggiorna gli <i>access time</i> (vedi sez. 5.2.4)
<code>MS_NODIRATIME</code>	2048	non aggiorna gli <i>access time</i> delle directory
<code>MS_BIND</code>	4096	monta il filesystem altrove
<code>MS_MOVE</code>	8192	sposta atomicamente il punto di montaggio

**Tabella 8.9:** Tabella dei codici dei flag di montaggio di un filesystem.

Per l'impostazione delle caratteristiche particolari di ciascun filesystem si usa invece l'argomento `data` che serve per passare le ulteriori informazioni necessarie, che ovviamente variano da filesystem a filesystem.

La funzione `mount` può essere utilizzata anche per effettuare il *rimontaggio* di un filesystem, cosa che permette di cambiarne al volo alcune delle caratteristiche di funzionamento (ad esempio passare da sola lettura a lettura/scrittura). Questa operazione è attivata attraverso uno dei bit di `mountflags`, `MS_REMOUNT`, che se impostato specifica che deve essere effettuato il rimontaggio del filesystem (con le opzioni specificate dagli altri bit), anche in questo caso il valore di `source` viene ignorato.

Una volta che non si voglia più utilizzare un certo filesystem è possibile *smontarlo* usando la funzione `umount`, il cui prototipo è:

```
#include <sys/mount.h>
umount(const char *target)
    Smonta il filesystem montato sulla directory target.
```

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

<code>EPERM</code>	il processo non ha i privilegi di amministratore.
<code>EBUSY</code>	<code>target</code> è la directory di lavoro di qualche processo, o contiene dei file aperti, o un altro mount point.
ed inoltre <code>ENOTDIR</code> , <code>EFAULT</code> , <code>ENOMEM</code> , <code>ENAMETOOLONG</code> , <code>ENOENT</code> o <code>ELOOP</code> .	

la funzione prende il nome della directory su cui il filesystem è montato e non il file o il dispositivo che è stato montato,<sup>9</sup> in quanto con il kernel 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 quando il filesystem è *occupato*, questo avviene quando ci sono ancora file aperti sul filesystem, se questo contiene la directory di lavoro corrente

<sup>8</sup>cioè un numero speciale usato come identificativo, che nel caso è 0xCOED; si può usare la costante `MS_MGC_MSK` per ottenere la parte di `mountflags` riservata al *magic number*.

<sup>9</sup>questo 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.

di un qualunque processo o il mount point di un altro filesystem; in questo caso l'errore restituito è **EBUSY**.

Linux provvede inoltre una seconda funzione, **umount2**, che in alcuni casi permette di forzare lo smontaggio di un filesystem, anche quando questo risulti occupato; il suo prototipo è:

```
#include <sys/mount.h>
umount2(const char *target, int flags)
    La funzione è identica a umount per comportamento e codici di errore, ma con flags si può
    specificare se forzare lo smontaggio.
```

Il valore di **flags** è una maschera binaria, e al momento l'unico valore definito è il bit **MNT\_FORCE**; gli altri bit devono essere nulli. 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.

Altre due funzioni specifiche di Linux,<sup>10</sup> 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)
    Restituisce in buf le informazioni relative al filesystem su cui è posto il file specificato.

Le funzioni ritornano 0 in caso di successo e -1 in caso di errore, nel qual caso errno assumerà
uno dei valori:
ENOSYS    il filesystem su cui si trova il file specificato non supporta la funzione.
e EFAULT ed EIO per entrambe, EBADF per fstatfs, ENOTDIR, ENAMETOOLONG, ENOENT, EACCES, ELOOP
per statfs.
```

Queste funzioni permettono di ottenere una serie di informazioni generali riguardo al filesystem su cui si trova il file specificato; queste vengono restituite all'indirizzo **buf** di una struttura **statfs** definita come in fig. 8.2, 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_spares[6];   /* riservati per uso futuro */
};
```

---

*Figura 8.2:* La struttura **statfs**.

Le *glibc* provvedono infine una serie di funzioni per la gestione dei due file **/etc/fstab** ed **/etc/mtab**, che convenzionalmente sono usati in quasi tutti i sistemi unix-like per mantenere

<sup>10</sup>esse si trovano anche su BSD, ma con una struttura diversa.

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, per `/etc/mtab` per inserire e rimuovere le voci presenti nel file.

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`, per cui ne tralasceremo la trattazione, rimandando al manuale delle *glibc* [3] per la documentazione completa.

### 8.2.3 La gestione delle informazioni su utenti e gruppi

Tradizionalmente le informazioni utilizzate nella gestione di utenti e gruppi (password, corrispondenze fra nomi simbolici e user-id, home directory, ecc.) venivano registrate all'interno dei due file di testo `/etc/passwd` ed `/etc/group`,<sup>11</sup> il cui formato è descritto dalle relative pagine del manuale<sup>12</sup> e tutte le funzioni che richiedevano l'accesso a queste informazione andavano a leggere direttamente il contenuto di questi file.

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,<sup>13</sup> svincolando completamente le singole applicazione dai dettagli del come questa viene eseguita e di dove vengono mantenuti i dati relativi; dall'altra con il diffondersi delle reti la necessità di centralizzare le informazioni degli utenti e dei gruppi per insiemi di macchine, in modo da mantenere coerenti i dati, ha portato anche alla necessità di poter recuperare e memorizzare dette informazioni su supporti diversi, introducendo il sistema del *Name Service Switch* che tratteremo brevemente più avanti (in sez. 16.1.1) dato che la maggior parte delle sue applicazioni sono relative alla 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. 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 mantenute. 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.

<sup>11</sup>in realtà oltre a questi nelle distribuzioni più recenti è stato introdotto il 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) per toglierle dagli altri file che devono poter essere letti per poter effettuare l'associazione fra username e `uid`.

<sup>12</sup>nella quinta sezione, quella dei file di configurazione, occorre cioè usare `man 5 passwd` dato che altrimenti si avrebbe la pagina di manuale del comando `passwd`.

<sup>13</sup>il *Pluggable Authentication Method* è 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 in maniera trasparente per le applicazioni purché per ciascun meccanismo si disponga della opportuna libreria che implementa l'interfaccia di PAM.

Le due funzioni forniscono le informazioni memorizzate nel registro degli utenti (che nelle versioni più recenti possono essere 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. 8.3, 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 8.3:* 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. Ovviamenete 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 un codice d'errore altrimenti, nel qual caso *errno* sarà impostata opportunamente.

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).

Del tutto analoghe alle precedenti sono le funzioni *getgrnam* e *getrgid* (e le relative analoghe rientranti con la stessa estensione *\_r*) che permettono di leggere le informazioni relative ai gruppi, i loro prototipi sono:

```
#include <grp.h>
#include <sys/types.h>
struct group *getrgid(gid_t gid)
struct group *getgrnam(const char *name)
struct group *getpwuid_r(gid_t gid, struct group *password, char *buffer, size_t
    buflen, struct group **result)
struct group *getpwnam_r(const char *name, struct group *password, char *buffer,
    size_t buflen, struct group **result)
    Restituiscono le informazioni relative al gruppo specificato.
```

Le funzioni ritornano 0 in caso di successo e un codice d'errore altrimenti, nel qual caso *errno* sarà impostata opportunamente.

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. 8.4.

---

```
struct group {
    char    *gr_name;           /* group name */
    char    *gr_passwd;         /* group password */
    gid_t   gr_gid;            /* group id */
    char    **gr_mem;           /* group members */
};
```

---

**Figura 8.4:** 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.<sup>14</sup> Dato che POSIX non prevede questa possibilità esiste un'altra interfaccia che lo fa, derivata da SVID le cui funzioni sono riportate in tab. 8.10. Questa 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 8.10:** Funzioni per la manipolazione dei campi di un file usato come registro per utenti o gruppi nel formato di `/etc/passwd` e `/etc/groups`.

Dato che oramai la gran parte delle distribuzioni di GNU/Linux utilizzano almeno 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, pertanto ci limiteremo a fornire soltanto l'elenco di tab. 8.10, senza nessuna spiegazione ulteriore. Chi volesse insistere ad usare questa interfaccia può fare riferimento alle pagine di manuale delle rispettive funzioni ed al manuale delle *glibc* per i dettagli del funzionamento.

<sup>14</sup>in realtà questo può essere fatto ricorrendo a PAM, ma questo è un altro discorso.

### 8.2.4 Il registro della **contabilità** degli utenti

L'ultimo 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<sup>15</sup> memorizzati nei due file `/var/run/utmp` e `/var/log/wtmp`. 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 delle *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 (vedi tab. 8.10) usate per accedere al registro degli utenti, 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, e a specificare su quale file esso viene mantenuto. I loro prototipi sono:

```
#include <utmp.h>
void utmpname(const char *file)
    Specifica il file da usare come registro.
void setutent(void)
    Apre il file del registro, posizionandosi al suo inizio.
void endutent(void)
    Chiude il file del registro.
```

Le funzioni non ritornano codici di errore.

In caso questo non venga specificato nessun file viene usato il valore standard `_PATH_UTMP` (che è definito in `paths.h`); in genere `utmpname` prevede due possibili valori:

`_PATH_UTMP` Specifica il registro per gli utenti correntemente collegati.

`_PATH_WTMP` Specifica il registro per l'archivio storico degli utenti collegati.

corrispondenti ai file `/var/run/utmp` e `/var/log/wtmp` visti in precedenza.

Una volta aperto il file si può eseguire una scansione leggendo o scrivendo una voce con le funzioni `getutent`, `getutid`, `getutline` e `pututline`, i cui prototipi sono:

---

<sup>15</sup>questa è la locazione specificata dal *Linux Filesystem Hierarchy Standard*, adottato dalla gran parte delle distribuzioni.

---

```

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 8.5:** La struttura `utmp` contenente le informazioni di una voce del registro di contabilità.

```

#include <utmp.h>
struct utmp *getutent(void)
    Legge una voce dalla posizione corrente nel registro.
struct utmp *getutid(struct utmp *ut)
    Ricerca una voce sul registro in base al contenuto di ut.
struct utmp *getutline(struct utmp *ut)
    Ricerca nel registro la prima voce corrispondente ad un processo sulla linea di terminale
    specificata tramite ut.
struct utmp *pututline(struct utmp *ut)
    Scrive una voce nel registro.

Le funzioni ritornano il puntatore ad una struttura utmp in caso di successo e NULL in caso di
errore.

```

Tutte queste funzioni fanno riferimento ad una struttura di tipo `utmp`, la cui definizione in Linux è riportata in fig. 8.5. 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.

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. 8.11, 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 8.11:** Classificazione delle voci del registro a seconda dei possibili valori del campo `ut_type`.

La funzione `getutline` esegue la ricerca sulle voci che hanno `ut_type` uguale a `LOGIN_PROCESS`

o `USER_PROCESS`, restituendo la prima che corrisponde al valore di `ut_line`, che specifica il device<sup>16</sup> di terminale che interessa. Lo stesso criterio di ricerca è usato da `pututline` per trovare uno spazio dove inserire la voce specificata, qualora non sia trovata 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 dalle *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 di `utmp`.

Le *glibc* utilizzano già una versione estesa di `utmp`, che rende inutili queste nuove strutture; pertanto esse e le relative funzioni di gestione (`getutxent`, `getutxid`, `getutxline`, `pututxline`, `setutxent` e `endutxent`) sono ridefinite come sinonimi delle funzioni appena viste.

Come visto in sez. 8.2.3, l'uso di strutture allocate staticamente rende le funzioni di lettura non rientranti; per questo motivo le *glibc* forniscono anche delle versioni rientranti: `getutent_r`, `getutid_r`, `getutline_r`, che invece di restituire un puntatore restituiscono un intero e prendono due argomenti aggiuntivi. Le funzioni si comportano esattamente come le analoghe non rientranti, solo che restituiscono il risultato all'indirizzo specificato dal primo argomento aggiuntivo (di tipo `struct utmp *buffer`) mentre il secondo (di tipo `struct utmp **result`) viene usato per restituire il puntatore allo stesso buffer.

Infine le *glibc* forniscono come estensione per la scrittura delle voci in `wtmp` altre due funzioni, `updwttmp` e `logwtmp`, i cui prototipi sono:

```
#include <utmp.h>
void updwttmp(const char *wtmp_file, const struct utmp *ut)
    Aggiunge la voce ut nel registro wtmp.
void logwtmp(const char *line, const char *name, const char *host)
    Aggiunge nel registro una voce con i valori specificati.
```

La prima funzione permette l'aggiunta di una voce a `wtmp` specificando direttamente una struttura `utmp`, mentre la seconda utilizza gli argomenti `line`, `name` e `host` per costruire la voce che poi aggiunge chiamando `updwttmp`.

## 8.3 Limitazione ed uso delle risorse

Dopo aver esaminato 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.

### 8.3.1 L'uso delle risorse

Come abbiamo accennato in sez. 3.2.6 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. 8.6.

La definizione della struttura in fig. 8.6 è ripresa da BSD 4.3,<sup>17</sup> ma attualmente (con i kernel della serie 2.4.x e 2.6.x) i soli campi che sono mantenuti sono: `ru_utime`, `ru_stime`, `ru_minflt`, `ru_majflt`, e `ru_nswap`. I primi due 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.

<sup>16</sup>espresso senza il `/dev/` iniziale.

<sup>17</sup>questo non ha a nulla a che fare con il *BSD accounting* 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.

---

```

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 8.6:** La struttura `rusage` per la lettura delle informazioni dei delle risorse usate da un processo.

Gli altri tre campi 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*), a quelli che invece han richiesto I/O su disco (detti invece *major page fault*) ed al numero di volte che il processo è stato completamente tolto dalla memoria per essere inserito nello swap.

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`.

Questa è la stessa struttura utilizzata da `wait4` (si ricordi quando visto in sez. 3.2.6) 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 `getrusage`, il cui prototipo è:

```

#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
int getrusage(int who, struct rusage *usage)
    Legge la quantità di risorse usate da un processo.

```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` può essere `EINVAL` o `EFAULT`.

L'argomento `who` permette di specificare il processo di cui si vuole leggere l'uso delle risorse; esso può assumere solo i due valori `RUSAGE_SELF` per indicare il processo corrente e `RUSAGE_CHILDREN` per indicare l'insieme dei processi figli di cui si è ricevuto lo stato di terminazione.

### 8.3.2 Limiti sulle risorse

Come accennato nell'introduzione oltre a mantenere i dati relativi all'uso delle risorse da parte dei vari processi, il kernel mette anche a disposizione delle funzioni con cui si possono impostare dei limiti sulle risorse che essi possono utilizzare. In generale ad ogni processo vengono associati due diversi limiti per ogni risorsa; questi sono detti il *limite corrente* (o *current limit*) che esprime il

valore massimo che attualmente il processo non può superare, ed il *limite massimo* (o *maximum limit*) che esprime il valore massimo che può assumere il *limite corrente*.

---

```
struct rlimit {
    rlim_t rlim_cur;      /* Soft limit */
    rlim_t rlim_max;     /* Hard limit (ceiling for rlim_cur) */

};
```

---

**Figura 8.7:** La struttura `rlimit` per impostare i limiti di utilizzo delle risorse usate da un processo.

In generale il primo viene chiamato anche *limite soffice* (o *soft limit*) dato che il suo valore può essere aumentato fino al valore del secondo, mentre il secondo è detto *limite duro* (o *hard limit*), in quanto un processo normale può solo diminuirne il valore. Il valore di questi due limiti è mantenuto in una struttura `rlimit`, la cui definizione è riportata in fig. 8.7, ed i cui campi corrispondono appunto a limite corrente e limite massimo.

In genere il superamento di un limite comporta o l'emissione di un segnale o il fallimento della system call che lo ha provocato; per permettere di leggere e di impostare i limiti di utilizzo delle risorse da parte di un processo Linux prevede due funzioni, `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)
    Legge il limite corrente per la risorsa resource.
int setrlimit(int resource, const struct rlimit *rlim)
    Imposta il limite per la risorsa resource.
```

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

**EINVAL** I valori per `resource` non sono validi.

**EPERM** Un processo senza i privilegi di amministratore ha cercato di innalzare i propri limiti.  
ed **EFAULT**.

Entrambe le funzioni permettono di specificare, attraverso l'argomento `resource`, su quale risorsa si vuole operare: i possibili valori di questo argomento sono elencati in tab. 8.12. L'accesso (rispettivamente in lettura e scrittura) ai valori effettivi dei limiti viene poi effettuato attraverso la struttura `rlimit` puntata da `rlim`.

Nello specificare un limite, oltre a fornire dei valori specifici, si può anche usare la costante `RLIM_INFINITY` che permette di sbloccare l'uso di una risorsa; ma si ricordi che solo un processo con i privilegi di amministratore può innalzare un limite al di sopra del valore corrente del limite massimo. Si tenga conto infine che tutti i limiti vengono ereditati dal processo padre attraverso una `fork` (vedi sez. 3.2.2) e mantenuti per gli altri programmi eseguiti attraverso una `exec` (vedi sez. 3.2.7).

### 8.3.3 Le 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.

<sup>18</sup>Impostare questo limite a zero è la maniera più semplice per evitare la creazione di `core` file (al proposito si veda sez. 9.2.2).

Valore	Significato
RLIMIT_CPU	Il massimo tempo di CPU che il processo può usare. Il superamento del limite comporta l'emissione di un segnale di <b>SIGXCPU</b> .
RLIMIT_FSIZE	La massima dimensione di un file che un processo può usare. Se il processo cerca di scrivere oltre questa dimensione riceverà un segnale di <b>SIGXFSZ</b> .
RLIMIT_DATA	La massima dimensione della memoria dati di un processo. Il tentativo di allocare più memoria causa il fallimento della funzione di allocazione.
RLIMIT_STACK	La massima dimensione dello stack del processo. Se il processo esegue operazioni che estendano lo stack oltre questa dimensione riceverà un segnale di <b>SIGSEGV</b> .
RLIMIT_CORE	La massima dimensione di un file di <i>core dump</i> creato da un processo. Nel caso le dimensioni dovessero essere maggiori il file non verrebbe generato. <sup>18</sup>
RLIMIT_RSS	L'ammontare massimo di memoria fisica dato al processo. Il limite è solo una indicazione per il kernel, qualora ci fosse un surplus di memoria questa verrebbe assegnata.
RLIMIT_NPROC	Il numero massimo di processi che possono essere creati sullo stesso user id. Se il limite viene raggiunto <b>fork</b> fallirà con un <b>EAGAIN</b> .
RLIMIT_NOFILE	Il numero massimo di file che il processo può aprire. L'apertura di un ulteriore file fallirà con un errore <b>EMFILE</b> .
RLIMIT_MEMLOCK	L'ammontare massimo di memoria che può essere bloccata in RAM senza paginazione (vedi sez. 2.2.7).
RLIMIT_AS	La dimensione massima di tutta la memoria che il processo può ottenere. Se il processo tenta di allocarne di più funzioni come <b>brk</b> , <b>malloc</b> o <b>mmap</b> falliranno.

**Tabella 8.12:** Valori possibili dell'argomento **resource** delle funzioni **getrlimit** e **setrlimit**.

In genere tutto ciò è del tutto trasparente al singolo processo, ma in certi casi, come per l'I/O mappato in memoria (vedi sez. 11.3.2) 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.

Di solito la dimensione delle pagine di memoria è fissata dall'architettura hardware, per cui il suo valore di norma veniva mantenuto in una costante che bastava utilizzare in fase di compilazione, ma oggi, con la presenza di alcune architetture (ad esempio Sun Sparc) che permettono di variare questa dimensione, per non dover ricompilare i programmi per ogni possibile modello e scelta di dimensioni, è necessario poter utilizzare una funzione.

Dato che si tratta di una caratteristica generale del sistema, questa dimensione può essere ottenuta come tutte le altre attraverso una chiamata a **sysconf** (nel caso **sysconf(\_SC\_PAGESIZE)**, ma in BSD 4.2 è stata introdotta una apposita funzione, **getpagesize**, che restituisce la dimensione delle pagine di memoria; il suo prototipo è:

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

Legge le dimensioni 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. 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, anche se le versioni delle librerie del C precedenti le *glibc* 2.1 implementavano questa funzione restituendo sempre un valore statico.

Le *glibc* forniscono, come specifica estensione GNU, altre due funzioni, **get\_phys\_pages** e **get\_avphys\_pages** che permettono di ottenere informazioni riguardo la memoria; i loro prototipi sono:

```
#include <sys/sysinfo.h>
long int get_phys_pages(void)
    Legge il numero totale di pagine di memoria disponibili per il sistema.
long int get_avphys_pages(void)
    Legge il numero di pagine di memoria disponibili nel sistema.

Le funzioni restituiscono un numero di pagine.
```

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.

Le *glibc* supportano 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 le *glibc* riprendono 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:

```
#include <stdlib.h>
int getloadavg(double loadavg[], int nelem)
    Legge il carico medio della macchina.

La funzione ritorna il numero di elementi scritti o -1 in caso di errore.
```

La funzione restituisce in ciascun elemento di `loadavg` il numero medio di processi attivi sulla coda dello scheduler, calcolato su un diverso 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.

## 8.4 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.

### 8.4.1 La misura del tempo in Unix

Storicamente i sistemi unix-like hanno sempre mantenuto due distinti tipi di dati per la misure dei tempi all'interno del sistema: essi sono rispettivamente chiamati *calendar time* e *process time*, secondo le definizioni:

#### *calendar time*

detto anche *tempo di calendario*. È il numero di secondi dalla mezzanotte del primo gennaio 1970, in tempo universale coordinato (o UTC), data che viene usualmente indicata con 00:00:00 Jan, 1 1970 (UTC) e chiamata *the Epoch*. Questo tempo viene anche chiamato anche GMT (Greenwich Mean Time) dato che l'UTC corrisponde all'ora locale di Greenwich. È il tempo su cui viene mantenuto l'orologio del kernel, e viene usato ad esempio per indicare le date di modifica dei file o quelle di avvio dei processi. Per memorizzare questo tempo è stato riservato il tipo primitivo `time_t`.

***process time***

detto talvolta *tempo di processore*. Viene misurato in *clock tick*. Un tempo questo corrispondeva al numero di interruzioni effettuate dal timer di sistema, adesso lo standard POSIX richiede che esso sia pari al valore 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.<sup>19</sup> Il dato primitivo usato per questo tempo è `clock_t`, che ha quindi una risoluzione del microsecondo. Il numero di tick al secondo può essere ricavato anche attraverso `sysconf` (vedi sez. 8.1.2). Il vecchio simbolo `CLK_TCK` definito in `time.h` è ormai considerato obsoleto.

In genere si usa il *calendar time* per esprimere le date dei file e le informazioni analoghe che riguardano i cosiddetti *tempi di orologio*, che vengono usati ad esempio per i demoni che compiono lavori amministrativi ad ore definite, come `cron`.

Di solito questo tempo viene convertito automaticamente dal valore in UTC al tempo locale, utilizzando le opportune informazioni di localizzazione (specificate in `/etc/timezone`). E da tenere presente che questo tempo è mantenuto dal sistema e non è detto che corrisponda al tempo tenuto dall'orologio hardware del calcolatore.

Anche il *process time* di solito si esprime in secondi, ma provvede una precisione ovviamente superiore al *calendar time* (che è mantenuto dal sistema con una granularità di un secondo) e viene usato per tenere conto dei tempi di esecuzione dei processi. Per ciascun processo il kernel calcola tre tempi diversi:

***clock time***

il tempo reale (viene chiamato anche *wall clock time* o *elapsed time*) passato dall'avvio del processo. Chiaramente tale tempo dipende anche dal carico del sistema e da quanti altri processi stavano girando nello stesso periodo.

***user time***

il tempo effettivo che il processore ha impiegato nell'esecuzione delle istruzioni del processo in user space. È quello riportato nella risorsa `ru_utime` di `rusage` vista in sez. 8.3.1.

***system time***

il tempo effettivo che il processore ha impiegato per eseguire codice delle system call nel kernel per conto del processo. È quello riportato nella risorsa `ru_stime` di `rusage` vista in sez. 8.3.1.

In genere la somma di *user time* e *system time* indica il tempo di processore totale che il sistema ha effettivamente utilizzato per eseguire un certo processo, questo viene chiamato anche *CPU time* o *tempo di CPU*. Si può ottenere un riassunto dei valori di questi tempi quando si esegue un qualsiasi programma lanciando quest'ultimo come argomento del comando `time`.

### 8.4.2 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:

---

<sup>19</sup>quest'ultima, come accennato in sez. 3.1.1, è invece data dalla costante `HZ`.

```
#include <time.h>
clock_t clock(void)
Legge il valore corrente del tempo di CPU.
```

La funzione ritorna il tempo di CPU usato dal programma e -1 in caso di errore.

La funzione restituisce il tempo in tick, quindi se si vuole il tempo in secondi occorre dividere il risultato per la costante `CLOCKS_PER_SEC`.<sup>20</sup> 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.

Come accennato in sez. 8.4.1 il tempo di CPU 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 attraverso la funzione `times`, il cui prototipo è:

```
#include <sys/times.h>
clock_t times(struct tms *buf)
Legge in buf il valore corrente dei tempi di processore.
```

La funzione ritorna il numero di clock tick dall'avvio del sistema in caso di successo e -1 in caso di errore.

La funzione restituisce i valori di process time del processo corrente in una struttura di tipo `tms`, la cui definizione è riportata in fig. 8.8. 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. 8.4.1.

---

```
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 8.8:** La struttura `tms` dei tempi di processore associati a un processo.

Gli altri due campi mantengono rispettivamente la somma dell'*user time* ed del *system time* di tutti i processi figli che sono terminati; il kernel cioè somma in `tms_cutime` il valore di `tms_utime` e `tms_cstime` per ciascun figlio del quale è stato ricevuto lo stato di terminazione, e lo stesso vale per `tms_cstime`.

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.

### 8.4.3 Le funzioni per il *calendar time*

Come anticipato in sez. 8.4.1 il *calendar time* è mantenuto dal kernel in 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). 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:

<sup>20</sup>le glibc seguono lo standard ANSI C, POSIX richiede che `CLOCKS_PER_SEC` sia definito pari a 1000000 indipendentemente dalla risoluzione del timer di sistema.

```
#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 in caso di errore, che può essere solo **EFAULT**.

dove t, se non nullo, deve essere l'indirizzo di una variabile su cui duplicare il valore di ritorno.

Analogamente la funzione **stime** 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 a t il valore corrente del calendar time.
```

La funzione ritorna 0 in caso di successo e -1 in caso di errore, che può essere **EFAULT** o **EPERM**.

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, altrimenti la chiamata fallirà con un errore di **EPERM**.

Data la scarsa precisione nell'uso di **time\_t** (che ha una risoluzione massima di un secondo) quando si devono effettuare operazioni sui tempi di norma l'uso delle funzioni precedenti è sconsigliato, ed esse sono di solito sostituite da **gettimeofday** e **settimeofday**,<sup>21</sup> 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.
```

Entrambe le funzioni restituiscono 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** può assumere i valori **EINVAL** **EFAULT** e per **settimeofday** anche **EPERM**.

Queste funzioni utilizzano una struttura di tipo **timeval**, la cui definizione, insieme a quella della analogia **timespec**, è riportata in fig. 8.9. Le glibc infatti forniscono queste due rappresentazioni alternative del *calendar time* che rispetto a **time\_t** consentono rispettivamente precisioni del microsecondo e del nanosecondo.<sup>22</sup>

---

```
struct timeval
{
    long tv_sec;           /* seconds */
    long tv_usec;          /* microseconds */
};

struct timespec {
    time_t   tv_sec;       /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

---

**Figura 8.9:** Le strutture **timeval** e **timespec** usate per una rappresentazione ad alta risoluzione del *calendar time*.

Come nel caso di **stime** anche **settimeofday** (la cosa continua a valere per qualunque funzio-

<sup>21</sup>le 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.

<sup>22</sup>la precisione è solo teorica, la precisione reale della misura del tempo dell'orologio di sistema non dipende dall'uso di queste strutture.

ne che vada a modificare l'orologio di sistema, quindi anche per quelle che tratteremo in seguito) può essere utilizzata solo da un processo coi privilegi di amministratore.

Il secondo parametro di entrambe le funzioni è una struttura `timezone`, che storicamente veniva utilizzata per specificare appunto la *time zone*, cioè l'insieme del fuso orario e delle convenzioni per l'ora legale che permettevano il passaggio dal tempo universale all'ora locale. Questo parametro oggi è obsoleto ed in Linux non è mai stato utilizzato; esso non è supportato né dalle vecchie `libc5`, né dalle `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:

```
#include <sys/time.h>
int adjtime(const struct timeval *delta, struct timeval *olddelta)
    Aggiusta del valore delta l'orologio di sistema.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà il valore `EPERM`.

Questa funzione permette di avere un aggiustamento graduale del tempo di sistema in modo che esso sia sempre crescente in maniera monotona. Il valore di `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. Il secondo parametro viene usato, se non nullo, per ricevere il valore dell'ultimo aggiustamento effettuato.

---

```
struct timex {
    unsigned int modes;      /* mode selector */
    long int offset;         /* time offset (usec) */
    long int freq;           /* frequency offset (scaled ppm) */
    long int maxerror;       /* maximum error (usec) */
    long int esterror;       /* estimated error (usec) */
    int status;              /* clock command/status */
    long int constant;       /* pll time constant */
    long int precision;      /* clock precision (usec) (read only) */
    long int tolerance;      /* clock frequency tolerance (ppm) (read only) */
    struct timeval time;     /* (read only) */
    long int tick;            /* (modified) usecs between clock ticks */
    long int ppsfreq;        /* pps frequency (scaled ppm) (ro) */
    long int jitter;          /* pps jitter (us) (ro) */
    int shift;                /* interval duration (s) (shift) (ro) */
    long int stabil;          /* pps stability (scaled ppm) (ro) */
    long int jitcnt;          /* jitter limit exceeded (ro) */
    long int calcnt;          /* calibration intervals (ro) */
    long int errcnt;          /* calibration errors (ro) */
    long int stbcnt;          /* stability limit exceeded (ro) */
};
```

---

**Figura 8.10:** La struttura `timex` per il controllo dell'orologio di sistema.

Linux poi prevede un'altra funzione, che consente un aggiustamento molto più dettagliato del

tempo, permettendo ad esempio anche di modificare anche la velocità dell'orologio di sistema. La funzione è `adjtimex` ed il suo prototipo è:

```
#include <sys/timex.h>
int adjtimex(struct timex *buf)
    Aggiusta del valore delta l'orologio di sistema.
```

La funzione restituisce lo stato dell'orologio (un valore > 0) in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori `EFAULT`, `EINVAL` ed `EPERM`.

La funzione richiede una struttura di tipo `timex`, la cui definizione, così come effettuata in `sys/timex.h`, è riportata in fig. 8.10. L'azione della funzione dipende dal valore del campo `mode`, che specifica quale parametro dell'orologio di sistema, specificato in un opportuno 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. 8.13.

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, le *glibc* provvedono 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. 8.13 i principali valori utilizzabili per il campo `mode`, un elenco più dettagliato del significato dei vari campi della struttura `timex` può essere ritrovato in [3].

Nome	Valore	Significato
ADJ_OFFSET	0x0001	Imposta la differenza fra il tempo reale e l'orologio di sistema, che deve essere indicata in microsecondi nel campo <code>offset</code> di <code>timex</code> .
ADJ_FREQUENCY	0x0002	Imposta la differenze in frequenza fra il tempo reale e l'orologio di sistema, che deve essere indicata in parti per milione nel campo <code>frequency</code> di <code>timex</code> .
ADJ_MAXERROR	0x0004	Imposta il valore massimo dell'errore sul tempo, espresso in microsecondi nel campo <code>maxerror</code> di <code>timex</code> .
ADJ_ESTERROR	0x0008	Imposta la stima dell'errore sul tempo, espresso in microsecondi nel campo <code>esterror</code> di <code>timex</code> .
ADJ_STATUS	0x0010	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	0x0020	Imposta la larghezza di banda del PLL implementato dal kernel, specificato nel campo <code>constant</code> di <code>timex</code> .
ADJ_TICK	0x4000	Imposta il valore dei tick del timer in microsecondi, espresso nel campo <code>tick</code> di <code>timex</code> .
ADJ_OFFSET_SINGLESHOT	0x8001	Imposta uno spostamento una tantum dell'orologio secondo il valore del campo <code>offset</code> simulando il comportamento di <code>adjtime</code> .

**Tabella 8.13:** Costanti per l'assegnazione del valore del campo `mode` della struttura `timex`.

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`.

La funzione ritorna un valore positivo che esprime lo stato dell'orologio di sistema; questo può assumere i valori riportati in tab. 8.14. Un valore di -1 viene usato per riportare un errore; al solito se si cercherà di modificare l'orologio di sistema (specificando un `mode` diverso da zero) senza avere i privilegi di amministratore si otterrà un errore di `EPERM`.

Nome	Valore	Significato
TIME_OK	0	L'orologio è sincronizzato.
TIME_INS	1	insert leap second.
TIME_DEL	2	delete leap second.
TIME_OOP	3	leap second in progress.
TIME_WAIT	4	leap second has occurred.
TIME_BAD	5	L'orologio non è sincronizzato.

**Tabella 8.14:** Possibili valori di ritorno di `adjtimex`.

#### 8.4.4 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.

Questo viene effettuato attraverso una opportuna struttura `tm`, la cui definizione è riportata in fig. 8.11, 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 trattare la gestione del fuso orario e dell'ora legale.<sup>23</sup>

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, i loro prototipi sono:

```
#include <time.h>
char *asctime(const struct tm *tm)
    Produce una stringa con data e ora partendo da un valore espresso in broken-down time.
char *ctime(const time_t *timep)
    Produce una stringa con data e ora partendo da un valore espresso in in formato time_t.
struct tm *gmtime(const time_t *timep)
    Converte il calendar time dato in formato time_t in un broken-down time espresso in UTC.
struct tm *localtime(const time_t *timep)
    Converte il calendar time dato in formato time_t in un broken-down time espresso nell'ora
    locale.
time_t mktime(struct tm *tm)
    Converte il broken-down time in formato time_t.
```

Tutte le funzioni restituiscono un puntatore al risultato in caso di successo e `NULL` in caso di errore, tranne che `mktme` che restituisce direttamente il valore o `-1` in caso di errore.

Le prime due funzioni, `asctime` e `ctime` servono per poter stampare in forma leggibile un tempo; esse restituiscono il puntatore ad una stringa, allocata staticamente, nella forma:

"Wed Jun 30 21:49:08 1993\n"

e impostano anche la variabile `tzname` con l'informazione della *time zone* corrente; `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 appendendo un `_r`, che prendono un secondo parametro `char *buf`, in cui l'utente deve specificare il buffer su cui la stringa deve essere copiata (deve essere di almeno 26 caratteri).

Le altre tre funzioni, `gmtime`, `localtime` e `mktme` servono per convertire il tempo dal formato `time_t` a quello di `tm` e viceversa; `gmtime` effettua la conversione usando il tempo coordinato

<sup>23</sup>in realtà i due campi `tm_gmtoff` e `tm_zone` sono estensioni previste da BSD e dalle *glibc*, che, quando è definita `_BSD_SOURCE`, hanno la forma in fig. 8.11.

---

```

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 8.11:** La struttura `tm` per una rappresentazione del tempo in termini di ora, minuti, secondi, ecc.

universale (UTC), cioè l'ora di Greenwich; mentre `localtime` usa l'ora locale; `mktime` esegue la conversione inversa.

Anche in questo caso le prime 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 parametro `struct tm *result`, fornito dal chiamante, che deve preallocare la struttura su cui sarà restituita la conversione.

Come mostrato in fig. 8.11 il *broken-down time* permette di tenere conto anche della differenza fra tempo universale e ora locale, compresa l'eventuale ora legale. Questo viene fatto attraverso le tre variabili globali mostrate in fig. 8.12, cui si accede quando si include `time.h`. Queste variabili vengono impostate quando si chiama una delle precedenti funzioni di conversione, oppure invocando direttamente la funzione `tzset`, il cui prototipo è:

```
#include <sys/timex.h>
void tzset(void)
    Imposta le variabili globali della time zone.
```

La funzione non ritorna niente e non dà errori.

La funzione inizializza le variabili di fig. 8.12 a partire dal valore della variabile di ambiente `TZ`, se quest'ultima non è definita verrà usato il file `/etc/localtime`.

---

```

extern char *tzname[2];
extern long timezone;
extern int daylight;
```

---

**Figura 8.12:** Le variabili globali usate per la gestione delle *time zone*.

La variabile `tzname` contiene due stringhe, che indicano i due nomi standard della *time zone* corrente. La prima è il nome per l'ora solare, la seconda per l'ora legale.<sup>24</sup> La variabile `timezone` indica la differenza di fuso orario in secondi, mentre `daylight` indica se è attiva o meno l'ora legale.

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:

<sup>24</sup>anche se sono indicati come `char *` non è il caso di modificare queste stringhe.

```
#include <time.h>
size_t strftime(char *s, size_t max, const char *format, const struct tm *tm)
    Stampa il tempo tm nella stringa s secondo il formato format.
```

La funzione ritorna il numero di caratteri stampati in **s**, altrimenti restituisce 0.

La funzione converte opportunamente il tempo **tm** in una stringa di testo da salvare in **s**, purché essa sia di dimensione, indicata da **size**, sufficiente. I caratteri generati dalla funzione vengono restituiti come valore di ritorno, ma non tengono 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.

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 8.15:** Valori previsti dallo standard ANSI C per modificatore della stringa di formato di **strftime**.

Il risultato della funzione è controllato dalla stringa di formato **format**, tutti i caratteri restano invariati eccetto % che viene utilizzato come modificatore; alcuni<sup>25</sup> dei possibili valori che esso può assumere sono riportati in tab. 8.15. La funzione tiene conto anche della presenza di una localizzazione per stampare in maniera adeguata i vari nomi.

## 8.5 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. 9) 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 basato sull'architettura a processi, e presenta una serie di problemi nel caso lo si debba usare con i thread.

<sup>25</sup>per la precisione quelli definiti dallo standard ANSI C, che sono anche quelli riportati da POSIX.1; le *glibc* provvedono tutte 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.

### 8.5.1 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 è -1 o un puntatore nullo o la costante `EOF` (a seconda della funzione); ma questo valore segnala solo che c'è stato un errore, non il tipo di errore.

Per riportare il tipo di errore il sistema usa la variabile globale `errno`,<sup>26</sup> definita nell'header `errno.h`; la variabile è in genere definita come `volatile` dato che può essere cambiata in modo asincrono da un segnale (si veda sez. 9.3.6 per un esempio, ricordando quanto trattato in sez. 3.5.2), ma dato che un gestore di segnale scritto bene salva e ripristina il valore della variabile, di questo non è necessario preoccuparsi nella programmazione normale.

I valori che può assumere `errno` sono riportati in app. C, nell'header `errno.h` sono anche definiti i nomi simbolici per le costanti numeriche che identificano i vari errori; essi iniziano tutti per E e si possono considerare come nomi riservati. In seguito faremo sempre riferimento a tali valori, 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, 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 ha successo, può chiamarne altre al suo interno che falliscono, modificando così `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 da seguire è sempre quella di controllare `errno` immediatamente dopo aver verificato il fallimento della funzione attraverso il suo codice di ritorno.

### 8.5.2 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 il messaggio di errore relativo ad errnum.
    La funzione ritorna il puntatore ad una stringa di errore.
```

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, e `errno` verrà modificata assumendo il valore `EINVAL`.

In generale `strerror` viene usata passando `errno` come parametro, 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 utilizza una stringa statica che non deve essere modificata dal programma; essa è utilizzabile solo fino ad una chiamata successiva a `strerror` o `perror`, nessun'altra funzione

<sup>26</sup>L'uso di una variabile globale può comportare alcuni problemi (ad esempio nel caso dei thread) ma lo standard ISO C consente anche di definire `errno` come un *modifiable lvalue*, quindi si può anche usare una macro, e questo è infatti il modo usato da Linux per renderla locale ai singoli thread.

di libreria tocca questa stringa. In ogni caso l'uso di una stringa statica rende la funzione non rientrante, per cui nel caso nel caso si usino i thread le librerie forniscono<sup>27</sup> 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 il messaggio di errore relativo ad errnum.
```

La funzione restituisce l'indirizzo del messaggio in caso di successo e **NULL** in caso di errore; nel qual caso **errno** assumerà i valori:

<b>EINVAL</b>	si è specificato un valore di <b>errnum</b> non valido.
<b>ERANGE</b>	la lunghezza di <b>buf</b> è insufficiente a contenere la stringa di errore.

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 viene troncata.

Una seconda funzione usata per riportare i codici di errore in maniera automatizzata sullo standard error (vedi sez. 6.1.2) è **perror**, il cui prototipo è:

```
#include <stdio.h>
void perror(const char *message)
    Stampa il messaggio di errore relativo al valore corrente di errno sullo standard error;
    preceduto dalla stringa message.
```

I messaggi di errore stampati sono gli stessi di **strerror**, (riportati in app. C), e, usando il valore corrente di **errno**, si riferiscono all'ultimo errore avvenuto. La stringa specificata con **message** viene stampato prima del messaggio d'errore, 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 questa stringa è sostanzialmente equivalente a quello di **strerror**.

In fig. 8.13 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 parametro 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.

### 8.5.3 Alcune estensioni GNU

Le precedenti funzioni sono quelle definite ed usate nei vari standard; le *glibc* hanno però introdotto una serie di estensioni “GNU” che forniscono alcune funzionalità aggiuntive per una gestione degli errori semplificata e più efficiente.

<sup>27</sup>questa funzione è la versione prevista dalle *glibc*, ed effettivamente definita in **string.h**, ne esiste una analoga nello standard SUSv3 (quella riportata dalla pagina di manuale), che restituisce **int** al posto di **char \***, e che tronca la stringa restituita a **size**.

---

```

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 8.13:** Codice per la stampa del messaggio di errore standard.

La prima estensione consiste in due variabili, `char * program_invocation_name` e `char * program_invocation_short_name` servono per ricavare il nome del programma; queste sono utili quando si deve aggiungere il nome del programma (cosa comune quando si ha un programma che non viene lanciato da linea di comando e salva gli errori in un file di log) al messaggio d'errore. La prima contiene il nome usato per lanciare il programma (ed è equivalente ad `argv[0]`); la seconda mantiene solo il nome del programma (senza eventuali directory in testa).

Uno dei problemi che si hanno con l'uso di `perror` è 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 maggiore informazione; 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 questo le *glibc* definiscono la funzione `error`, il cui prototipo è:

```
#include <stdio.h>
void error(int status, int errnum, const char *format, ...)
    Stampa un messaggio di errore formattato.
```

La funzione non restituisce nulla e non riporta errori.

La funzione fa parte delle estensioni GNU per la gestione degli errori, l'argomento `format` prende la stessa sintassi di `printf`, ed i relativi parametri 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 `errnum`, 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 errnum, const char *fname, unsigned int
    lineno, const char *format, ...)
    Stampa un messaggio di errore formattato.
```

La funzione non restituisce 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 9

## 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, etc.) 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 seconda di uso e modalità di generazione fino ad esaminare in dettaglio funzioni e le metodologie di gestione avanzate e le estensioni fatte all'interfaccia classica nelle nuove versioni dello standard POSIX.

### 9.1 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.

#### 9.1.1 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 di terminare o fermare il programma. In genere si realizza attraverso un segnale mandato dalla shell in corrispondenza della pressione di tasti del terminale come **C-c** o **C-z**.<sup>1</sup>
- l'esecuzione di una **kill** o di una **raise** da parte del processo stesso o di un'altro (solo nel caso della **kill**).

---

<sup>1</sup>indichiamo con **C-x** la pressione simultanea al tasto **x** del tasto control (ctrl in molte tastiere).

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 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 routine di gestione (quello che da qui in avanti chiameremo il gestore del segnale, dall'inglese *signal handler*) che può essere stata specificata dall'utente (nel qual caso si dice che si *intercetta* il segnale).

### 9.1.2 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* (quella implementata dalle prime versioni di Unix) la routine di gestione del segnale specificata dall'utente non resta attiva una volta che è stata eseguita; è perciò compito dell'utente stesso ripetere l'installazione all'interno del gestore del segnale, in tutti quei casi in cui si vuole che esso resti attivo.

In questo caso è possibile una situazione in cui i segnali possono essere perduti. Si consideri il segmento di codice riportato in fig. 9.1, nel programma principale viene installato un gestore (5), ed in quest'ultimo la prima operazione (11) è quella di reinstallare se stesso. Se nell'esecuzione del gestore un secondo segnale arriva prima che esso abbia potuto eseguire la reinstallazione, verrà eseguito il comportamento predefinito assegnato al segnale stesso, il che può comportare, a seconda dei casi, che il segnale viene perso (se l'impostazione predefinita era quello di ignorarlo) o la terminazione immediata del processo; in entrambi i casi l'azione prevista non verrà eseguita.

---

```

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     ...
13 }
```

---

**Figura 9.1:** Esempio di codice di un gestore di segnale per la semantica inaffidabile.

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 (sull'argomento vedi quanto detto in sez. 3.5).

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 l'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. 9.4.4) per determinare quali segnali sono bloccati e quali sono pendenti.

### 9.1.3 Tipi di segnali

In generale gli eventi che generano segnali si possono dividere 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 la condizione di errore più comune comporta la restituzione di un codice di errore da parte di una funzione di libreria, sono gli errori che possono avvenire ovunque in un programma che causano l'emissione di un segnale, come le divisioni per zero o l'uso di indirizzi di memoria non validi.

Un evento esterno ha in genere a che fare con l'I/O o con altri processi; esempi di segnali di questo tipo sono quelli legati all'arrivo di dati di input, scadenze di un timer, terminazione di processi figli.

Una richiesta esplicita significa l'uso di una chiamata di sistema (come `kill` o `raise`) per la generazione di un segnale, cosa che viene fatta usualmente dalla shell quando l'utente invoca la sequenza di tasti di stop o di suspend, ma può essere pure inserita all'interno di un programma.

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.

### 9.1.4 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 il segnale *pendente* sarà subito notificato. Si tenga presente però che 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.

Si ricordi però 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 è 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 (`SIGKILL` e `SIGSTOP`) questa azione è fissa e non può essere cambiata, ma per tutti gli altri si può selezionare una delle tre possibilità seguenti:

- ignorare il segnale.
- catturare 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` (vedi sez. 9.3.2 e sez. 9.4.3). 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 race condition).

Nel caso non sia stata specificata un'azione, viene utilizzata l'azione standard che (come vedremo in sez. 9.2.1) è propria di ciascun segnale; nella maggior parte dei casi essa porta alla terminazione del processo, ma alcuni segnali che rappresentano eventi innocui vengono ignorati.

Quando un segnale termina un processo, il padre può determinare la causa della terminazione esaminando il codice di stato riportato delle funzioni `wait` e `waitpid` (vedi sez. 3.2.5); questo 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 anche la caratteristica di scrivere un file di *core dump* che registra lo stato del processo (ed in particolare della memoria e dello stack) prima della terminazione. Questo può essere esaminato in seguito con un debugger per investigare sulla causa dell'errore. Lo stesso avviene se i suddetti segnale vengono generati con una `kill`.

## 9.2 La classificazione dei segnali

Esamineremo in questa sezione quali sono i vari segnali definiti nel sistema, le loro caratteristiche e tipologia, le varie macro e costanti che permettono di identificarli, e le funzioni che ne stampano la descrizione.

### 9.2.1 I segnali standard

Ciascun segnale è identificato rispetto al sistema da un numero, ma l'uso diretto di questo numero da parte dei programmi è da evitare, in quanto esso può variare a seconda dell'implementazione

del sistema, e nel caso di Linux, anche a seconda dell'architettura hardware. Per questo motivo ad ogni segnale viene associato un nome, definendo con una macro di preprocessore una costante uguale al suddetto numero. Sono questi nomi, che sono standardizzati e sostanzialmente uniformi rispetto alle varie implementazioni, che si devono usare nei programmi. Tutti i nomi e le funzioni che concernevano i segnali sono definiti nell'header di sistema `signal.h`.

Il numero totale di segnali presenti è dato dalla macro `NSIG`, e dato che i numeri dei segnali sono allocati progressivamente, essa corrisponde anche al successivo del valore numerico assegnato all'ultimo segnale definito. In tab. 9.3 si è riportato l'elenco completo dei segnali definiti in Linux (estratto dalle pagine di manuale), comparati con quelli definiti in vari standard.

Sigla	Significato
A	L'azione predefinita è terminare il processo.
B	L'azione predefinita è ignorare il segnale.
C	L'azione predefinita è terminare il processo e scrivere un <i>core dump</i> .
D	L'azione predefinita è fermare il processo.
E	Il segnale non può essere intercettato.
F	Il segnale non può essere ignorato.

**Tabella 9.1:** Legenda delle azioni predefinite dei segnali riportate in tab. 9.3.

In tab. 9.3 si sono anche riportate le azioni predefinite di ciascun segnale (riassunte con delle lettere, la cui legenda completa è in tab. 9.1), quando nessun gestore è installato un segnale può essere ignorato o causare la terminazione del processo. Nella colonna **Standard** sono stati indicati anche gli standard in cui ciascun segnale è definito, secondo lo schema di tab. 9.2.

Sigla	Standard
P	POSIX.
B	BSD.
L	Linux.
S	SUSv2.

**Tabella 9.2:** Legenda dei valori della colonna **Standard** di tab. 9.3.

In alcuni casi alla terminazione del processo è associata la creazione di un file (posto nella directory corrente del processo e chiamato `core`) su cui viene salvata un'immagine della memoria del processo (il cosiddetto *core dump*), che può essere usata da un debugger per esaminare lo stato dello stack e delle variabili al momento della ricezione del segnale.

La descrizione dettagliata del significato dei vari segnali, raggruppati per tipologia, verrà affrontate nei paragrafi successivi.

### 9.2.2 Segnali di errore di programma

Questi segnali sono generati quando il sistema, o in certi casi direttamente l'hardware (come per i *page fault* non validi) 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

Segnale	Standard	Azione	Descrizione
SIGHUP	PL	A	Hangup o terminazione del processo di controllo
SIGINT	PL	A	Interrupt da tastiera (C-c)
SIGQUIT	PL	C	Quit da tastiera (C-y)
SIGILL	PL	C	Istruzione illecita
SIGABRT	PL	C	Segnale di abort da <code>abort</code>
SIGFPE	PL	C	Errore aritmetico
SIGKILL	PL	AEF	Segnale di terminazione forzata
SIGSEGV	PL	C	Errore di accesso in memoria
SIGPIPE	PL	A	Pipe spezzata
SIGALRM	PL	A	Segnale del timer da <code>alarm</code>
SIGTERM	PL	A	Segnale di terminazione C-\
SIGUSR1	PL	A	Segnale utente numero 1
SIGUSR2	PL	A	Segnale utente numero 2
SIGCHLD	PL	B	Figlio terminato o fermato
SIGCONT	PL		Continua se fermato
SIGSTOP	PL	DEF	Ferma il processo
SIGTSTP	PL	D	Pressione del tasto di stop sul terminale
SIGTTIN	PL	D	Input sul terminale per un processo in background
SIGTTOU	PL	D	Output sul terminale per un processo in background
SIGBUS	SL	C	Errore sul bus (bad memory access)
SIGPOLL	SL	A	<i>Pollable event</i> (Sys V). Sinonimo di <b>SIGIO</b>
SIGPROF	SL	A	Timer del profiling scaduto
SIGSYS	SL	C	Argomento sbagliato per una subroutine (SVID)
SIGTRAP	SL	C	Trappole per un Trace/breakpoint
SIGURG	SLB	B	Ricezione di una <i>urgent condition</i> su un socket
SIGVTALRM	SLB	A	Virtual alarm clock
SIGXCPU	SLB	C	Ecceduto il limite sul CPU time
SIGXFSZ	SLB	C	Ecceduto il limite sulla dimensione dei file
SIGIOT	L	C	IOT trap. Sinonimo di <b>SIGABRT</b>
SIGEMT	L		
SIGSTKFLT	L	A	Errore sullo stack del coprocessore
SIGIO	LB	A	L'I/O è possibile (4.2 BSD)
SIGCLD	L		Sinonimo di <b>SIGCHLD</b>
SIGPWR	L	A	Fallimento dell'alimentazione
SIGINFO	L		Sinonimo di <b>SIGPWR</b>
SIGLOST	L	A	Perso un lock sul file (per NFS)
SIGWINCH	LB	B	Finestra ridimensionata (4.3 BSD, Sun)
SIGUNUSED	L	A	Segnale inutilizzato (diventerà <b>SIGSYS</b> )

**Tabella 9.3:** Lista dei segnali in Linux.

*core dump* che viene scritto in un file `core` nella directory corrente del processo al momento dell'errore, che il 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.

<b>SIGSEGV</b>	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. Se il gestore ritorna il comportamento del processo è indefinito. È tipico ottenere questo segnale dereferenziando un puntatore nullo o non inizializzato leggendo al di là della fine di un vettore.
<b>SIGBUS</b>	Il nome deriva da <i>bus error</i> . Come <b>SIGSEGV</b> questo è un segnale che viene generato di solito quando si dereferenzia un puntatore non inizializzato, la differenza è che <b>SIGSEGV</b> indica un accesso non permesso su un indirizzo esistente (tipo fuori dallo heap o dallo stack), mentre <b>SIGBUS</b> indica l'accesso ad un indirizzo non valido, come nel caso di un puntatore non allineato.
<b>SIGABRT</b>	Il nome deriva da <i>abort</i> . Il segnale indica che il programma stesso ha rilevato un errore che viene riportato chiamando la funzione <b>abort</b> che genera questo segnale.
<b>SIGTRAP</b>	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 se un programma normale non dovrebbe ricevere questo segnale.
<b>SIGSYS</b>	Sta ad indicare che si è eseguita una istruzione che richiede l'esecuzione di una system call, ma si è fornito un codice sbagliato per quest'ultima.

### 9.2.3 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 trattare 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:

<b>SIGTERM</b>	Il nome sta per <i>terminate</i> . È un segnale generico usato per causare la conclusione di un programma. Al contrario di <b>SIGKILL</b> può essere intercettato, ignorato, bloccato. In genere lo si usa per chiedere in maniera "educata" ad un processo di concludersi.
<b>SIGINT</b>	Il nome sta per <i>interrupt</i> . È il segnale di interruzione per il programma. È quello che viene generato di default dal comando <b>kill</b> o dall'invio sul terminale del carattere di controllo INTR (interrupt, generato dalla sequenza C-c).
<b>SIGQUIT</b>	È analogo a <b>SIGINT</b> con le differenze che è controllato da un altro carattere di controllo, QUIT, corrispondente alla sequenza C-\. 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, in modo che essi possano disconnettersi dal relativo terminale.

Viene inoltre usato in genere per segnalare ai demoni (che non hanno un terminale di controllo) la necessità di reinizializzarsi e rileggere il/i file di configurazione.

#### 9.2.4 I segnali di allarme

Questi segnali sono generati dalla scadenza di un timer. 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`.

**SIGVTALRM** 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.

#### 9.2.5 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 `fcntl` abbia avuto successo.

**SIGURG** Questo segnale è inviato quando arrivano dei dati urgenti o *out-of-band* su di un socket; per maggiori dettagli al proposito si veda sez. 18.1.1.

**SIGPOLL** Questo segnale è equivalente a **SIGIO**, è definito solo per compatibilità con i sistemi System V.

### 9.2.6 I segnali per il controllo di sessione

Questi sono i segnali usati dal controllo delle sessioni e dei processi, il loro uso è specifico 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. 3.2.5.   |
| SIGCLD  | Per Linux questo è solo un segnale identico al precedente, il nome è obsoleto e andrebbe evitato.  |
| SIGCONT | Il nome sta per <i>continue</i> . Il segnale viene usato per fare ripartire un programma precedentemente fermato da <b>SIGSTOP</b> . 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.<br><br>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. |
| SIGSTOP | Il segnale ferma un processo (lo porta cioè in uno stato di sleep, vedi sez. 3.4.1); 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 SUSP (prodotto dalla combinazione C-z), ed al contrario di <b>SIGSTOP</b> 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'echo 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 background 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. 10.1.1.  |
| SIGTTOU | Segnale analogo al precedente <b>SIGTTIN</b> , ma generato quando si tenta di scrivere o modificare uno dei modi del terminale. L'azione predefinita è di fermare il processo, l'argomento è trattato in sez. 10.1.1.  |

### 9.2.7 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 di terminare il processo, questi segnali sono:

- |         |   |
|---------|---|
| SIGPIPE | Sta per <i>Broken pipe</i> . 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. 12.1.1). 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`.

SIGLOST	Sta per <i>Resource lost</i> . Tradizionalmente è il segnale che generato quando si ha un advisory lock su un file su NFS che viene perso 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 <code>SIGIO</code> . <sup>2</sup>
SIGXCPU	Sta per <i>CPU time limit exceeded</i> . Questo segnale è generato quando un processo eccede il limite impostato per il tempo di CPU disponibile, vedi sez. 8.3.2.
SIGXFSZ	Sta per <i>File size limit exceeded</i> . 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. 8.3.2.

### 9.2.8 Ulteriori segnali

Raccogliamo qui infine una serie di segnali che hanno scopi differenti non classificabili in maniera omogenea. Questi segnali sono:

SIGUSR1	Insieme a <code>SIGUSR2</code> è un segnale a disposizione dell'utente che lo può usare per quello che vuole. Viene generato solo attraverso l'invocazione della funzione <code>kill</code> . 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. Vedi quanto appena detto per <code>SIGUSR1</code> .
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.

### 9.2.9 Le funzioni `strsignal` e `psignal`

Per la descrizione dei segnali il sistema mette a disposizione due funzioni che stampano un messaggio di descrizione dato il numero. In genere si usano quando si vuole notificare all'utente il segnale ricevuto (nel caso di terminazione di un processo figlio o di un gestore che gestisce più segnali); la prima funzione, `strsignal`, è una estensione GNU, accessibile avendo definito `_GNU_SOURCE`, ed è analoga alla funzione `strerror` (si veda sez. 8.5.2) per gli errori:

```
#include <string.h>
char *strsignal(int signum)
Ritorna il puntatore ad una stringa che contiene la descrizione del segnale signum.
```

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.

<sup>2</sup>ed è segnalato come BUG nella pagina di manuale.

La seconda funzione, `psignal`, deriva da BSD ed è analoga alla funzione `perror` descritta sempre in sez. 8.5.2; il suo prototipo è:

```
#include <signal.h>
void psignal(int sig, const char *s)
    Stampa sullo standard error un messaggio costituito dalla stringa s, seguita da due punti
    ed una descrizione del segnale indicato da sig.
```

Una modalità alternativa per utilizzare le descrizioni restituite da `strsignal` e `psignal` è quello di fare usare la variabile `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]`.

## 9.3 La gestione 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 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.

### 9.3.1 Il comportamento generale del sistema.

Abbiamo già trattato in sez. 9.1 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. 3.2.2 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. 9.4.4). 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. 3.2.7) 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 (che per questo motivo vengono chiamate *lente*) possono bloccarsi indefiniteamente. 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.
- le funzioni di intercomunicazione che si bloccano in attesa di risposte da altri processi.
- la funzione `pause` (usata appunto per attendere l'arrivo di un segnale).
- la funzione `wait` (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 le *glibc* provvedono 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 le *glibc* consentono di utilizzare entrambi gli approcci, attraverso una opportuna opzione di `sigaction` (vedi sez. 9.4.3). 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.

### 9.3.2 La funzione `signal`

L'interfaccia più semplice per la gestione dei segnali è costituita dalla funzione `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; è questo il motivo per cui ogni implementazione successiva ne ha modificato e ridefinito il comportamento, pur mantenendone immutato il prototipo<sup>3</sup> che è:

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler)
```

Installa la funzione di gestione `handler` (il gestore) per il segnale `signum`.

La funzione ritorna il precedente gestore in caso di successo o `SIG_ERR` in caso di errore.

---

<sup>3</sup>in realtà in alcune vecchie implementazioni (SVr4 e 4.3+BSD in particolare) vengono usati alcuni parametri aggiuntivi per definire il comportamento della funzione, vedremo in sez. 9.4.3 che questo è possibile usando la funzione `sigaction`.

In questa definizione si è usato un tipo di dato, `sighandler_t`, che è una estensione GNU, definita dalle *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 poca chiarezza 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) e che prende un argomento di tipo `int`.<sup>4</sup> La funzione `signal` quindi restituisce e prende come secondo argomento un puntatore a una funzione di questo tipo, che è appunto il gestore del segnale.

Il numero di segnale passato in `signum` può essere indicato direttamente con una delle costanti definite in sez. 9.2.1. Il gestore `handler` invece, oltre all'indirizzo della funzione da chiamare all'occorrenza del segnale, può assumere anche i due valori costanti `SIG_IGN` con cui si dice ignorare il segnale e `SIG_DFL` per reinstallare l'azione predefinita.<sup>5</sup>

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 un `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 le *libc4* e le *libc5*.<sup>6</sup>

Al contrario BSD segue la semantica affidabile, non disinstallando il gestore e bloccando il segnale durante l'esecuzione dello stesso. Con l'utilizzo delle *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. 9.1.2, può essere ottenuto chiamando `sysv_signal`, una volta che si sia definita la macro `_XOPEN_SOURCE`. In generale, per evitare questi problemi, l'uso di `signal` (ed ogni eventuale ridefinizione della stessa) è da evitare; tutti i nuovi programmi dovrebbero usare `sigaction`.

È da tenere presente che, 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 ad una `raise`) è indefinito. Un gestore che ritorna da questi segnali può dare luogo ad un ciclo infinito.

### 9.3.3 Le funzioni `kill` e `raise`

Come accennato in sez. 9.1.3, un segnale può essere generato direttamente da un processo attraverso una opportuna system call. Le funzioni che si usano di solito per inviare un segnale generico sono due, `raise` e `kill`.

---

<sup>4</sup>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`.

<sup>5</sup>si ricordi però che i due segnali `SIGKILL` e `SIGSTOP` non possono essere ignorati né intercettati; l'uso di `SIG_IGN` per questi segnali non ha alcun effetto.

<sup>6</sup>nelle *libc5* esiste però la possibilità di includere `bsd/signal.h` al posto di `signal.h`, nel qual caso la funzione `signal` viene ridefinita per seguire la semantica affidabile usata da BSD.

La prima funzione è `raise`, che è definita dallo standard ANSI C, e serve per inviare un segnale al processo corrente,<sup>7</sup> il suo prototipo è:

```
#include <signal.h>
int raise(int sig)
    Invia il segnale sig al processo corrente.
```

La funzione restituisce zero in caso di successo e -1 per un errore, il solo errore restituito è `EINVAL` qualora si sia specificato un numero di segnale invalido.

Il valore di `sig` specifica il segnale che si vuole inviare e può essere specificato con una delle macro definite in sez. 9.2. 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`.

Mentre `raise` è una funzione di libreria, quando si vuole inviare un segnale generico ad un processo occorre utilizzare la apposita system call, questa può essere chiamata attraverso la funzione `kill`, il cui prototipo è:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig)
    Invia il segnale sig al processo specificato con pid.
```

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

<code>EINVAL</code>	Il segnale specificato non esiste.
<code>ESRCH</code>	Il processo selezionato non esiste.
<code>EPERM</code>	Non si hanno privilegi sufficienti ad inviare il segnale.

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 specificato non esiste. Si tenga conto però che il sistema ricicla i `pid` (come accennato in sez. 3.2.1) per cui l'esistenza di un processo non significa che esso sia realmente quello a cui si intendeva mandare il segnale.

Il valore dell'argomento `pid` specifica il processo (o i processi) di destinazione a cui il segnale deve essere inviato e può assumere i valori riportati in tab. 9.4.

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 per essere più portabile.

Una seconda funzione che può essere definita in termini di `kill` è `killpg`, che è sostanzialmente equivalente a `kill(-pidgrp, signal)`; il suo prototipo è:

```
#include <signal.h>
int killpg(pid_t pidgrp, int signal)
    Invia il segnale signal al process group pidgrp.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, gli errori sono gli stessi di `kill`.

e che permette di inviare un segnale a tutto un *process group* (vedi sez. 10.1.2).

Solo l'amministratore può inviare un segnale ad un processo qualunque, in tutti gli altri casi l'user-ID reale o l'user-ID effettivo del processo chiamante devono corrispondere all'user-ID reale

<sup>7</sup>non prevedendo la presenza di un sistema multutente 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.

Valore	Significato
> 0	il segnale è mandato al processo con il <i>pid</i> indicato.
0	il segnale è mandato ad ogni processo del <i>process group</i> del chiamante.
-1	il segnale è mandato ad ogni processo (eccetto <b>init</b> ).
< -1	il segnale è mandato ad ogni processo del process group <b> pid </b> .

**Tabella 9.4:** Valori dell'argomento *pid* per la funzione **kill**.

o all'user-ID salvato della destinazione. Fa eccezione il caso in cui il segnale inviato sia **SIGCONT**, nel quale occorre che entrambi i processi appartengano alla stessa sessione. Inoltre, dato il ruolo fondamentale che riveste nel sistema (si ricordi quanto visto in sez. 9.2.3), 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.

### 9.3.4 Le funzioni **alarm** e **abort**

Un caso particolare di segnali generati a richiesta è quello che riguarda i vari segnali di temporizzazione e **SIGABRT**, per ciascuno di questi segnali sono previste funzioni specifiche che ne effettuino l'invio. La più comune delle funzioni usate per la temporizzazione è **alarm** il cui prototipo è:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
    Predispone l'invio di SIGALRM dopo seconds secondi.
```

La funzione restituisce il numero di secondi rimanenti ad un precedente allarme, o zero se non c'erano allarmi pendenti.

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 da **seconds**.

Se si specifica per **seconds** un valore nullo non verrà inviato nessun segnale; siccome alla chiamata viene cancellato ogni precedente allarme, questo può essere usato per cancellare una programmazione precedente.

La funzione inoltre ritorna il numero di secondi rimanenti all'invio dell'allarme precedentemente programmato, in modo che sia possibile controllare se non si cancella un precedente allarme ed eventualmente predisporre le opportune misure per gestire il caso di necessità di più interruzioni.

In sez. 8.4.1 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. 8.4.1 abbiamo chiamato *CPU 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)
    Predisponde l'invio di un segnale di allarme alla scadenza dell'intervalle value sul timer
    specificato da which.

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso errno assumerà
uno dei valori EINVAL o EFAULT.
```

Il valore di **which** permette di specificare quale dei tre timer illustrati in precedenza usare; i possibili valori sono riportati in tab. 9.5.

Valore	Timer
ITIMER_REAL	<i>real-time timer</i>
ITIMER_VIRTUAL	<i>virtual timer</i>
ITIMER_PROF	<i>profiling timer</i>

**Tabella 9.5:** 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. 5.5.

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 9.2:** 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 delle *glibc* [3] che ne riporta la definizione mostrata in fig. 9.3.

Si deve comunque tenere presente che la precisione di queste funzioni è limitata da quella della frequenza del timer di sistema (che nel caso dei PC significa circa 10 ms). Il sistema assicura comunque che il segnale non sarà mai generato prima della scadenza programmata (l'arrotondamento cioè è sempre effettuato per eccesso).

---

```

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 9.3:* Definizione di `alarm` in termini di `setitimer`.

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_VIRT`) la consegna è immediata, altrimenti può esserci un ulteriore ritardo che può variare a seconda del carico del sistema.

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. 9.3.6, un solo segnale sarà consegnato.

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 in value il valore del timer specificato da which.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore e restituisce gli stessi errori di `getitimer`

i cui parametri hanno lo stesso significato e formato di quelli di `setitimer`.

L'ultima funzione che permette l'invio diretto di un segnale è `abort`; che, come accennato in sez. 3.2.4, permette di abortire l'esecuzione di un programma tramite l'invio di `SIGABRT`. Il suo prototipo è:

```
#include <stdlib.h>
void abort(void)
    Abortisce il processo corrente.

La funzione non ritorna, il processo è terminato inviando il segnale di SIGABRT.
```

La differenza fra questa funzione e l'uso di `raise` 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 `at_exit` e `on_exit`.

### 9.3.5 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 opportuno loop di attesa, ma in un sistema multitasking un loop di attesa è solo un inutile spreco di CPU, per questo ci sono apposite funzioni che permettono di mettere un processo in stato di attesa.<sup>8</sup>

Il metodo tradizionale per fare attendere ad un processo fino all'arrivo di un segnale è quello di usare la funzione `pause`, il cui prototipo è:

```
#include <unistd.h>
int pause(void)
    Pone il processo in stato di sleep fino al ritorno di un gestore.
```

La funzione ritorna solo dopo che un segnale è stato ricevuto ed il relativo gestore è ritornato, nel qual caso restituisce -1 e `errno` assumerà il valore `EINTR`.

La funzione segnala sempre una condizione di errore (il successo sarebbe quello di 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 nello standard POSIX.1 viene definita la funzione `sleep`, il cui prototipo è:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds)
    Pone il processo in stato di sleep per seconds secondi.
```

La funzione restituisce zero se l'attesa viene completata, o il numero di secondi restanti se viene interrotta da un segnale.

La funzione attende per il tempo specificato, a meno di non essere interrotta da un segnale. In questo caso non è una buona idea ripetere la chiamata per il tempo rimanente, in quanto la riattivazione del processo può avvenire in un qualunque momento, ma il valore restituito sarà sempre arrotondato al secondo, con la conseguenza che, se la successione dei segnali è particolarmente sfortunata e le differenze si accumulano, si potranno avere ritardi anche di parecchi secondi. In genere la scelta più sicura è quella di stabilire un termine per l'attesa, e ricalcolare tutte le volte il numero di secondi da aspettare.

In alcune implementazioni inoltre 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 maniera analoga all'esempio che vedremo in sez. 9.4.1). In tal caso mescolare chiamata di `alarm` e `sleep` o modificare l'azione di `SIGALRM`, può causare risultati indefiniti. Nel caso delle *glibc* è stata usata una implementazione completamente indipendente e questi problemi non ci sono.

La granularità di `sleep` permette di specificare attese soltanto in secondi, per questo sia sotto BSD4.3 che in SUSv2 è stata definita la funzione `usleep` (dove la `u` è intesa come sostituzione di  $\mu$ ); i due standard hanno delle definizioni diverse, ma le *glibc* seguono<sup>9</sup> seguono quella di SUSv2 che prevede il seguente prototipo:

```
#include <unistd.h>
int usleep(unsigned long usec)
    Pone il processo in stato di sleep per usec microsecondi.
```

La funzione restituisce zero se l'attesa viene completata, o -1 in caso di errore, nel qual caso `errno` assumerà il valore `EINTR`.

<sup>8</sup>si tratta in sostanza di funzioni che permettono di portare esplicitamente il processo in stato di *sleep*, vedi sez. 3.4.1.

<sup>9</sup>secondo la pagina di manuale almeno dalla versione 2.2.2.

Anche questa funzione, a seconda delle implementazioni, può presentare problemi nell’interazione con `alarm` e `SIGALRM`. È pertanto deprecata in favore della funzione `nanosleep`, definita dallo standard POSIX1.b, il cui prototipo è:

```
#include <unistd.h>
int nanosleep(const struct timespec *req, struct timespec *rem)
    Pone il processo in stato di sleep per il tempo specificato da req. In caso di interruzione
    restituisce il tempo restante in rem.
```

La funzione restituisce zero se l’attesa viene completata, o -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EINVAL</code>	si è specificato un numero di secondi negativo o un numero di nanosecondi maggiore di 999.999.999.
<code>EINTR</code>	la funzione è stata interrotta da un segnale.

Lo standard richiede che la funzione sia implementata in maniera del tutto indipendente da `alarm`<sup>10</sup> e sia utilizzabile senza interferenze con l’uso di `SIGALRM`. La funzione prende come parametri delle strutture di tipo `timespec`, la cui definizione è riportata in fig. 8.9, che permettono 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, e basta richiamare la funzione per completare l’attesa.

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 il successivo giro di scheduler e 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.

In realtà è possibile ottenere anche pause più precise del centesimo di secondo usando politiche di scheduling real time come `SCHED_FIFO` o `SCHED_RR`; in tal caso infatti il meccanismo di scheduling ordinario viene evitato, e si raggiungono pause fino ai 2 ms con precisioni del  $\mu$ s.

### 9.3.6 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. 3.2.4 che una delle azioni eseguite dal kernel alla conclusione di un processo è quella di inviare questo segnale al padre.<sup>11</sup> 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 chiamare `waitpid` per completare la procedura di terminazione in modo da evitare la formazione di zombie.

In fig. 9.4 è mostrato il codice contenente una implementazione generica di una routine di gestione per `SIGCHLD`, (che si trova nei sorgenti allegati nel file `SigHand.c`); se ripetiamo i test di sez. 3.2.4, 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. 8.5.1) si comincia (12-13) con il salvare lo stato corrente di `errno`, in modo

<sup>10</sup>nel caso di Linux questo è fatto utilizzando direttamente il timer del kernel.

<sup>11</sup>in realtà in SVr4 eredita la semantica di System V, in cui il segnale si chiama `SIGCLD` e viene trattato in maniera speciale; in System V infatti se si imposta esplicitamente l’azione a `SIG_IGN` il segnale non viene generato ed il sistema non genera zombie (lo stato di terminazione viene scartato senza dover chiamare una `wait`). L’azione predefinita è sempre quella di ignorare il segnale, ma non attiva questo comportamento. Linux, come BSD e POSIX, non supporta questa semantica ed usa il nome di `SIGCLD` come sinonimo di `SIGCHLD`.

---

```

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         if (pid > 0) {
13             debug("child %d terminated with status %x\n", pid, status);
14         }
15     } while ((pid > 0) && (errno == EINTR));
16     /* restore errno value */
17     errno = errno_save;
18     /* return */
19     return;
20 }
```

---

**Figura 9.4:** Codice di una funzione generica di gestione per il segnale SIGCHLD.

da poterlo ripristinare prima del ritorno del gestore (22-23). In questo modo si preserva il valore della variabile visto dal corso di esecuzione principale del processo, che sarebbe altrimenti sarebbe sovrascritto dal valore restituito nella successiva chiamata di `wait`.

Il compito principale del gestore è quello di ricevere lo stato di terminazione del processo, cosa che viene eseguita nel ciclo in (15-21). 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 capitì 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 sarà 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. 3.2.5 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.

## 9.4 Gestione avanzata

Le funzioni esaminate finora fanno riferimento ad 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 evi-

denzi, 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.

#### 9.4.1 Alcune problematiche aperte

Come accennato in sez. 9.3.5 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. 9.5.

Dato che è nostra intenzione utilizzare `SIGALRM` il primo passo della nostra implementazione di 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 (17-19) fino alla sua ricezione. Al ritorno di `pause`, causato dal ritorno del gestore (1-9), si ripristina il gestore originario (20-21) restituendo l'eventuale tempo rimanente (22-23) che potrà essere diverso da zero qualora l'interruzione di `pause` venisse causata da un altro segnale.

---

```

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 9.5:* Una implementazione pericolosa di `sleep`.

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 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. 2.4.4) 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. 9.6.

---

```

1 static jmp_buf alarm_return;
2 unsigned int sleep(unsigned int seconds)
3 {
4     signandler_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 9.6:* Una implementazione ancora malfunzionante di `sleep`.

In questo caso il gestore (18-26) non ritorna come in fig. 9.5, ma usa `longjmp` (24) per rientrare nel corpo principale del programma; dato che in questo caso il valore di uscita di `setjmp` è 1, grazie alla condizione in (9-12) si evita 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, in questo caso l'esecuzione non riprenderà nel gestore in questione, ma nel ciclo principale, interrompendone inopportunamente l'esecuzione. Lo stesso tipo di problemi si presenterebbero se si volesse usare `alarm` per stabilire un timeout su una qualunque system call bloccante.

Un secondo esempio è quello in cui si usa il segnale per notificare una qualche forma di evento; in genere quello che si fa in questo caso è impostare nel gestore un opportuno flag da controllare nel corpo principale del programma (con un codice del tipo di quello riportato in fig. 9.7).

La logica è quella di far impostare al gestore (14-19) una variabile globale preventivamente inizializzata nel programma principale, il quale potrà determinare, osservandone il contenuto, l'occorrenza o meno del segnale, e prendere le relative azioni conseguenti (6-11).

Questo è il tipico esempio di caso, già citato in sez. 3.5.2, in cui si genera una race condition ; se infatti il segnale arriva immediatamente dopo l'esecuzione del controllo (6) ma prima della cancellazione del flag (7), la sua occorrenza sarà perduta.

Questi esempi ci mostrano che per una gestione effettiva dei segnali occorrono funzioni più sofisticate di quelle illustrate finora, che hanno origine dalla interfaccia semplice, ma poco sofisticata, dei primi sistemi Unix, in modo da consentire la gestione di tutti i possibili aspetti con cui un processo deve reagire alla ricezione di un segnale.

---

```

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 9.7:** Un esempio non funzionante del codice per il controllo di un evento generato da un segnale.

#### 9.4.2 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 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), che è il tipo di dato che viene usato per gestire il blocco dei segnali.

In genere un *insieme di segnali* è rappresentato da un intero di dimensione opportuna, di solito si pari al numero di bit dell'architettura della macchina<sup>12</sup>, ciascun bit del quale è associato ad uno specifico segnale; in questo modo è di solito possibile implementare le operazioni direttamente con istruzioni elementari del processore; lo standard POSIX.1 definisce cinque funzioni per la manipolazione degli insiemi di segnali: **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset** e **sigismember**, i cui prototipi sono:

#include <signal.h>	
int sigemptyset(sigset_t *set)	Inizializza un insieme di segnali vuoto (in cui non c'è nessun segnale).
int sigfillset(sigset_t *set)	Inizializza un insieme di segnali pieno (in cui ci sono tutti i segnali).
int sigaddset(sigset_t *set, int signum)	Aggiunge il segnale <b>signum</b> all'insieme di segnali <b>set</b> .
int sigdelset(sigset_t *set, int signum)	Toglie il segnale <b>signum</b> dall'insieme di segnali <b>set</b> .
int sigismember(const sigset_t *set, int signum)	Controlla se il segnale <b>signum</b> è nell'insieme di segnali <b>set</b> .

Le prime quattro funzioni ritornano 0 in caso di successo, mentre **sigismember** ritorna 1 se **signum** è in **set** e 0 altrimenti. In caso di errore tutte ritornano -1, con **errno** impostata a **EINVAL** (il solo errore possibile è che **signum** non sia un segnale valido).

<sup>12</sup>nel caso dei PC questo comporta un massimo di 32 segnali distinti, dato che in Linux questi sono sufficienti non c'è necessità di nessuna struttura più complicata.

Dato che in generale non si può fare conto sulle caratteristiche di una implementazione (non è detto che si disponga di un numero di bit sufficienti per mettere tutti i segnali in un intero, o in `sigset_t` possono essere immagazzinate ulteriori informazioni) tutte le operazioni devono essere comunque eseguite attraverso queste funzioni.

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. 9.4.4). Essi possono essere definiti in due diverse maniere, aggiungendo i segnali voluti ad un insieme vuoto ottenuto con `sigemptyset` o togliendo quelli che non servono da un insieme completo ottenuto con `sigfillset`. Infine `sigismember` permette di verificare la presenza di uno specifico segnale in un insieme.

#### 9.4.3 La funzione `sigaction`

Abbiamo già accennato in sez. 9.3.2 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 funzione principale dell'interfaccia POSIX.1 per i 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 il segnale signum.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<b>EINVAL</b>	Si è specificato un numero di segnale invalido o si è cercato di installare il gestore per SIGKILL o SIGSTOP.
<b>EFAULT</b>	Si sono specificati indirizzi non validi.

La funzione serve ad installare una nuova azione per il segnale `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 lo standard raccomanda di usare sempre questa funzione al posto di `signal` (che in genere viene definita tramite essa), in quanto 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.

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 dallo standard POSIX.1 ed in Linux è definita secondo quanto riportato in fig. 9.8. 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 comunque la maschera dei segnali bloccati (vedi sez. 9.4.4) viene 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. 9.6. 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

---

```
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 9.8:** La struttura `sigaction`.

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. 9.6.

Valore	Significato
<code>SA_NOCLDSTOP</code>	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>SIGTTOUT</code> .
<code>SA_ONESHOT</code>	Ristabilisce l'azione per il segnale al valore predefinito una volta che il gestore è stato lanciato, riproduce cioè il comportamento della semantica inaffidabile.
<code>SA_RESETHAND</code>	Sinonimo di <code>SA_ONESHOT</code> .
<code>SA_RESTART</code>	Riavvia automaticamente le <i>slow system call</i> quando vengono interrotte dal suddetto segnale; riproduce cioè il comportamento standard di BSD.
<code>SA_NOMASK</code>	Evita che il segnale corrente sia bloccato durante l'esecuzione del gestore.
<code>SA_NODEFER</code>	Sinonimo di <code>SA_NOMASK</code> .
<code>SA_SIGINFO</code>	Deve essere specificato quando si vuole usare un gestore in forma estesa usando <code>sa_sigaction</code> al posto di <code>sa_handler</code> .
<code>SA_ONSTACK</code>	Stabilisce l'uso di uno stack alternativo per l'esecuzione del gestore (vedi sez. 9.4.5).

**Tabella 9.6:** Valori del campo `sa_flag` della struttura `sigaction`.

Come si può notare in fig. 9.8 `sigaction` permette<sup>13</sup> di utilizzare due forme diverse di gestore, da specificare, a seconda dell'uso o meno del flag `SA_SIGINFO`, rispettivamente attraverso i campi `sa_sigaction` o `sa_handler`,<sup>14</sup> 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. 9.9.

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.

In generale `si_code` contiene, per i segnali generici, per quelli real-time e per tutti quelli inviati tramite `kill`, informazioni circa l'origine del segnale (se generato dal kernel, da un timer,

<sup>13</sup>La 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. 9.4.6). In precedenza era possibile ottenere alcune informazioni addizionali usando `sa_handler` con un secondo parametro addizionale di tipo `sigcontext`, che adesso è deprecato.

<sup>14</sup>i due tipi devono essere usati in maniera alternativa, in certe implementazioni questi campi vengono addirittura definiti come `union`.

---

```

siginfo_t {
    int      si_signo; /* Signal number */
    int      si_errno; /* An errno value */
    int      si_code; /* Signal code */
    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 */
    void *   si_addr; /* Memory location which caused fault */
    int      si_band; /* Band event */
    int      si_fd; /* File descriptor */
}

```

---

*Figura 9.9:* La struttura `siginfo_t`.

da `kill`, ecc.). Alcuni segnali però usano `si_code` per fornire una informazione specifica: ad esempio i vari segnali di errore (`SIGFPE`, `SIGILL`, `SIGBUS` e `SIGSEGV`) lo usano per fornire maggiori dettagli riguardo l'errore (come il tipo di errore aritmetico, di istruzione illegittima o di violazione di memoria) mentre alcuni segnali di controllo (`SIGCHLD`, `SIGTRAP` e `SIGPOLL`) forniscono altre informazioni specifiche. In tutti i casi il valore del campo è riportato attraverso delle costanti (le cui definizioni si trovano `bits/siginfo.h`) il cui elenco dettagliato è disponibile nella pagina di manuale di `sigaction`.

Il resto della struttura è definito come `union` ed i valori eventualmente presenti dipendono dal segnale, così `SIGCHLD` ed i segnali real-time (vedi sez. 9.4.6) inviati tramite `kill` avvalorano `si_pid` e `si_uid` coi valori corrispondenti al processo che ha emesso il segnale, `SIGILL`, `SIGFPE`, `SIGSEGV` e `SIGBUS` avvalorano `si_addr` con l'indirizzo cui è avvenuto l'errore, `SIGIO` (vedi sez. 11.2.2) avvalora `si_fd` con il numero del file descriptor e `si_band` per i dati urgenti su un socket.

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`.

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, il cui codice è riportato in fig. 9.10 (il codice completo si trova nel file `SigHand.c` nei sorgenti allegati). Si noti come, essendo la funzione estremamente semplice, è definita come `inline`.<sup>15</sup>

<sup>15</sup>la direttiva `inline` 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 (impostando 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

---

```

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 9.10:* La funzione `Signal`, equivalente a `signal`, definita attraverso `sigaction`.

#### 9.4.4 La gestione della *maschera dei segnali* o *signal mask*

Come spiegato in sez. 9.1.2 tutti i moderni sistemi unix-like permettono di bloccare temporaneamente (o di eliminare completamente, impostando `SIG_IGN` come azione) la consegna dei segnali ad un processo. Questo è fatto specificando la cosiddetta *maschera dei segnali* (o *signal mask*) del processo<sup>16</sup> cioè l'insieme dei segnali la cui consegna è bloccata. Abbiamo accennato in sez. 3.2.2 che la *signal mask* 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, attraverso l'uso del campo `sa_mask` di `sigaction`.

Uno dei problemi evidenziatisi con l'esempio di fig. 9.7 è che in molti casi è necessario proteggere delle sezioni di codice (nel caso in questione la sezione fra il controllo e la eventuale cancellazione del flag che testimoniava l'avvenuta occorrenza del segnale) in modo da essere sicuri che essi siano eseguiti senza interruzioni.

Le operazioni più semplici, come l'assegnazione o il controllo di una variabile (per essere sicuri si può usare il tipo `sig_atomic_t`) di norma sono atomiche, quando occorrono operazioni più complesse si può invece usare la funzione `sigprocmask` che permette di bloccare uno o più segnali; il suo prototipo è:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
    Cambia la maschera dei segnali del processo corrente.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<code>EINVAL</code>	Si è specificato un numero di segnale invalido.
<code>EFAULT</code>	Si sono specificati indirizzi non validi.

La funzione usa l'insieme di segnali dato all'indirizzo `set` per modificare la maschera dei segnali del processo corrente. La modifica viene effettuata a seconda del valore dell'argomento

---

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 [8]) che in questo modo possono essere evitati.

<sup>16</sup>nel caso di Linux essa è mantenuta dal campo `blocked` della `task_struct` del processo.

how, secondo le modalità specificate in tab. 9.7. Qualora si specifichi un valore non nullo per oldset la maschera dei segnali corrente viene salvata a quell'indirizzo.

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 9.7:** Valori e significato dell'argomento how della funzione `sigprocmask`.

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. 9.7, proteggendo la sezione fra il controllo del flag e la sua cancellazione.

La funzione può essere usata anche all'interno di un gestore, ad 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 alla conclusione del terminatore.

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. 9.6, 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 `sigsuspend`, il cui prototipo è:

```
#include <signal.h>
int sigsuspend(const sigset_t *mask)
    Imposta la signal mask specificata, mettendo in attesa il processo.

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso errno assumerà i
valori:
EINVAL    Si è specificato un numero di segnale invalido.
EFAULT    Si sono specificati indirizzi non validi.
```

Come esempio dell'uso di queste funzioni proviamo a riscrivere un'altra volta l'esempio di implementazione di `sleep`. Abbiamo accennato in sez. 9.4.3 come con `sigaction` sia possibile bloccare `SIGALRM` nell'installazione dei gestori degli altri segnali, per poter usare l'implementazione vista in fig. 9.6 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. 9.11 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. 9.6 evitando l'uso di `longjmp`. Come in precedenza il gestore (35-37) non esegue nessuna operazione, limitandosi a ritornare per interrompere il programma messo in attesa.

La prima parte della funzione (11-15) provvede ad installare l'opportuno gestore per `SIGALRM`, salvando quello originario, che sarà ripristinato alla conclusione della stessa (28); il passo successivo è quello di bloccare `SIGALRM` (17-19) per evitare che esso possa essere ricevuto dal processo fra l'esecuzione di `alarm` (21) e la sospensione dello stesso. Nel fare questo si salva la maschera corrente dei segnali, che sarà ripristinata alla fine (27), e al contempo si prepara la maschera dei segnali `sleep_mask` per riattivare `SIGALRM` all'esecuzione di `sigsuspend`.

---

```

1 void alarm_hand(int);
2 unsigned int sleep(unsigned int seconds)
3 {
4     struct sigaction new_action, old_action;
5     sigset(SIG_BLOCK, 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 setsuspend */
30 }

```

---

*Figura 9.11:* Una implementazione completa di `sleep`.

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`.
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`.

#### 9.4.5 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 `sigpending`, anch’essa introdotta dallo standard POSIX.1; il suo prototipo è:

```
#include <signal.h>
int sigpending(sigset_t *set)
```

Scrive in `set` l’insieme dei segnali pendenti.

La funzione restituisce zero in caso di successo e -1 per un errore.

La funzione permette di ricavare quali sono i segnali pendenti per il processo in corso, cioè i segnali che sono stato 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. 9.6) 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 un nuovo stack per i segnali.
```

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

<code>ENOMEM</code>	La dimensione specificata per il nuovo stack è minore di <code>MINSIGSTKSZ</code> .
<code>EPERM</code>	Uno degli indirizzi non è valido.
<code>EFAULT</code>	Si è cercato di cambiare lo stack alternativo mentre questo è attivo (cioè il processo è in esecuzione su di esso).
<code>EINVAL</code>	<code>ss</code> non è nullo e <code>ss_flags</code> contiene un valore diverso da zero che non è <code>SS_DISABLE</code> .

La funzione prende come argomenti puntatori ad una struttura di tipo `stack_t`, definita in fig. 9.12. 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).

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

---

```

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 9.12:* La struttura `stack_t`.

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 imposto con chiamata de 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. 9.6 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 BSD prevede che sia ripristinata la maschera dei segnali precedente l'invocazione, come per un normale ritorno, mentre System V no.

Lo standard POSIX.1 non specifica questo comportamento per `setjmp` e `longjmp`, ed il comportamento delle *glibc* dipende da quale delle caratteristiche si sono abilitate con le macro viste in sez. 1.2.8.

Lo standard POSIX però prevede anche la presenza di altre due funzioni `sigsetjmp` e `siglongjmp`, che permettono di decidere 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 per un salto non-locale.
void siglongjmp(sigjmp_buf env, int val)
    Esegue un salto non-locale su un precedente contesto.

```

Le due funzioni sono identiche alle analoghe `setjmp` e `longjmp` di sez. 2.4.4, ma consentono di specificare il comportamento sul ripristino o meno della maschera dei segnali.

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. 2.4.4 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 sarà salvata in `env` e ripristinata in un successivo `siglongjmp`; quest'ultima funzione, a parte l'uso di `sigjmp_buf` per `env`, è assolutamente identica a `longjmp`.

#### 9.4.6 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,<sup>17</sup> 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 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`.

Queste nuove funzionalità (eccetto l'ultima, che, come vedremo, è parzialmente disponibile anche con i segnali ordinari) si applicano solo ai nuovi segnali real-time; questi ultimi sono accessibili in un range di valori specificati dalle due macro `SIGRTMIN` e `SIGRTMAX`,<sup>18</sup> che specificano il numero minimo e massimo associato ad un segnale real-time.

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.

Si tenga presente che questi nuovi segnali non sono associati a nessun evento specifico, a meno di non utilizzarli in meccanismi di notifica come quelli per l'I/O asincrono (vedi sez. 11.2.2) o per le code di messaggi POSIX (vedi sez. 12.4.2); 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

<sup>17</sup> questa estensione è stata introdotta in Linux a partire dal kernel 2.1.43(?), e dalle *glibc* 2.1(?).

<sup>18</sup> in Linux di solito il primo valore è 32, ed il secondo `_NSIG-1`, che di norma è 63, per un totale di 32 segnali disponibili, contro gli almeno 8 richiesti da POSIX.1b.

utilizzare la forma estesa `sa_sigaction` (vedi sez. 9.4.3). 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 abbiamo già visto in fig. 9.9, 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'user-ID effettivo del processo che ha inviato il segnale, mentre per la restituzione dei dati viene usato il campo `si_value`.

Questo è una `union` di tipo `sigval_t` (la sua definizione è in fig. 9.13) in cui può essere memorizzato o un valore numerico, se usata nella forma `sival_int`, o un indirizzo, se usata nella forma `sival_ptr`. L'unione viene usata dai segnali real-time e da vari meccanismi di notifica<sup>19</sup> per restituire dati al gestore del segnale; in alcune definizioni essa viene identificata anche come `union sigval`.

---

```
union sigval_t {
    int sival_int;
    void *sival_ptr;
}
```

---

*Figura 9.13:* La unione `sigval_t`.

A causa delle loro caratteristiche, la funzione `kill` non è adatta ad inviare segnali real-time, poichè non è in grado di fornire alcun valore per `sigval_t`; 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 sigval_t value)
    Invia il segnale signo al processo pid, restituendo al gestore il valore value.
```

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

<code>EAGAIN</code>	La coda è esaurita, ci sono già <code>SIGQUEUE_MAX</code> segnali in attesa si consegna.
<code>EPERM</code>	Non si hanno privilegi appropriati per inviare il segnale al processo specificato.
<code>ESRCH</code>	Il processo <code>pid</code> non esiste.
<code>EINVAL</code>	Si è specificato un valore non valido per <code>signo</code> .
ed inoltre <code>ENOMEM</code> .	

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<sup>20</sup> nella coda dei segnali real-time) esso viene inserito e diventa pendente; una volta consegnato riporterà nel campo `si_code` di `siginfo_t` il valore `SI_QUEUE` e il campo `si_value` riceverà quanto inviato con `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.

Lo standard POSIX.1b definisce inoltre delle nuove funzioni che permettono di gestire l'attesa di segnali specifici su una coda, esse servono in particolar modo nel caso dei thread, in cui si

<sup>19</sup>un campo di tipo `sigval_t` è presente anche nella struttura `sigevent` che viene usata dai meccanismi di notifica come quelli per l'I/O asincrono (vedi sez. 11.2.2) o le code di messaggi POSIX (vedi sez. 12.4.2).

<sup>20</sup>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. 8.1.1. Il suo valore minimo secondo lo standard, `_POSIX_SIGQUEUE_MAX`, è pari a 32. Nel caso di Linux questo è uno dei parametri del kernel impostabili sia con `sysctl`, che scrivendolo direttamente in `/proc/sys/kernel/rtsig-max`, il valore predefinito è di 1024.

possono usare i segnali real-time come meccanismi di comunicazione elementare; la prima di queste funzioni è **sigwait**, il cui prototipo è:

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig)
    Attende che uno dei segnali specificati in set sia pendente.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà uno dei valori:

<b>EINTR</b>	La funzione è stata interrotta.
<b>EINVAL</b>	Si è specificato un valore non valido per <b>set</b> .
ed inoltre <b>EFAULT</b> .	

La funzione estrae dall'insieme dei segnali pendenti uno qualunque dei segnali specificati da **set**, il cui valore viene restituito in **sig**. Se sono pendenti più segnali, viene estratto quello a priorità più alta (cioè 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, 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)
    Analoga a sigwait, ma riceve anche le informazioni associate al segnale in info.
int sigtimedwait(const sigset_t *set, siginfo_t *value, const struct timespec
    *info)
    Analoga a sigwaitinfo, con un la possibilità di specificare un timeout in timeout.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà uno dei valori già visti per **sigwait**, ai quali si aggiunge, per **sigtimedwait**:

<b>EAGAIN</b>	Si è superato il timeout senza che un segnale atteso fosse emesso.
---------------	--

Entrambe le funzioni sono estensioni di **sigwait**. La prima permette di ricevere, oltre al numero del segnale, anche le informazioni ad esso associate tramite **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, scaduto il quale ritornerà con un errore. Se si specifica 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 com 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, per evitare che venga eseguita l'azione predefinita, i segnali gestiti in questa maniera devono essere mascherati per tutti i thread, compreso quello dedicato alla gestione, che potrebbe riceverlo fra due chiamate successive.

# Capitolo 10

## Terminali e sessioni di lavoro

I terminali per lungo tempo sono stati l'unico modo per accedere al sistema, per questo anche oggi che esistono molte altre interfacce, essi continuano a coprire un ruolo particolare, restando strettamente legati al funzionamento dell'interfaccia a linea di comando.

Nella prima parte del capitolo esamineremo i concetti base del sistema delle sessioni di lavoro, vale a dire il metodo con cui il kernel permette ad un utente di gestire le capacità multitasking del sistema, permettendo di eseguire più programmi in contemporanea. Nella seconda parte del capitolo tratteremo poi il funzionamento dell'I/O su terminale, e delle varie peculiarità che esso viene ad assumere a causa del suo stretto legame con il suo uso come interfaccia di accesso al sistema da parte degli utenti.

### 10.1 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,<sup>1</sup> avendo cioè un solo punto in cui si può avere accesso all'input ed all'output degli stessi.

#### 10.1.1 Una panoramica introduttiva

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 fanno), 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. 9.2.6.

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*, che riunisce (vedi sez. 10.1.2) tutti i processi eseguiti all'interno dello stesso login (esamineremo tutto il processo in dettaglio in sez. 10.1.4).

Siccome la shell è collegata ad un solo terminale, che viene usualmente chiamato *terminale di controllo*, (vedi sez. 10.1.3) un solo comando alla volta (quello che viene detto in *foreground*), potrà scrivere e leggere dal terminale. La shell però può eseguire anche più comandi in contemporanea, mandandoli in *background* (aggiungendo una `&` alla fine del comando), nel qual caso essi saranno eseguiti senza essere collegati al terminale.

---

<sup>1</sup>con X e con i terminali virtuali 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.

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 programmi in una sola riga di comando con le pipe, ed in tal caso verranno eseguiti più programmi, inoltre, anche quando si invoca un singolo programma, questo potrà sempre lanciare sottoprocessi per eseguire dei compiti specifici.

Per questo l'esecuzione di un 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 *process group* (detto anche *raggruppamento di processi*, vedi sez. 10.1.2) e la shell farà sì che tutti i processi che originano da una riga di comando appartengano allo stesso raggruppamento, in modo che le varie funzioni di controllo, ed i segnali inviati dal terminale, possano fare riferimento ad esso.

In generale allora 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. 10.1.3).

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**.<sup>2</sup> Il comando **fg** consente invece di mettere in *foreground* un comando precedentemente lanciato in *background*.

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 di **waitpid**, di rilevare sia i processi che sono terminati, sia i raggruppamenti che sono bloccati (in questo caso usando l'opzione **WUNTRACED**, secondo quanto illustrato in sez. 3.2.5).

### 10.1.2 I *process group* e le *sessions*

Come accennato in sez. 10.1.1 nel job control i processi vengono raggruppati in *process group* e *sessions*; per far questo vengono utilizzati due ulteriori identificatori (oltre quelli visti in sez. 3.2.1) che il kernel associa a ciascun processo:<sup>3</sup> 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 **getpgid** e **getpgrp**,<sup>4</sup> i cui prototipi sono:

```
#include <unistd.h>
pid_t getpgid(pid_t pid)
    Legge il pgid del processo pid.
pid_t getpgrp(void)
    Legge il pgid del processo corrente.
```

Le funzioni restituiscono il *pgid* del processo, **getpgrp** ha sempre successo, mentre **getpgid** restituisce -1 ponendo **errno** a **ESRCH** se il processo selezionato non esiste.

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; **getpgrp** è di norma equivalente a **getpgid(0)**.

<sup>2</sup>si tenga presente che **bg** e **fg** sono parole chiave che indicano comandi interni alla shell, e nel caso non comportano l'esecuzione di un programma esterno.

<sup>3</sup>in Linux questi identificatori sono mantenuti nei campi **pgrp** e **session** della struttura **task\_struct** definita in **sched.h**.

<sup>4</sup>**getpgrp** è definita nello standard POSIX.1, mentre **getpgid** è richiesta da SVr4.

In maniera analoga l'identificatore della sessione può essere letto dalla funzione `getsid`, che però nelle *glibc*<sup>5</sup> è accessibile solo definendo `_XOPEN_SOURCE` e `_XOPEN_SOURCE_EXTENDED`; il suo prototipo è:

```
#include <unistd.h>
pid_t getsid(pid_t pid)
    Legge l'identificatore di sessione del processo pid.
```

La funzione restituisce l'identificatore (un numero positivo) in caso di successo, e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

**ESRCH** Il processo selezionato non esiste.

**EPERM** In alcune implementazioni viene restituito quando il processo selezionato non fa parte della stessa sessione del processo corrente.

Entrambi gli identificatori vengono inizializzati alla 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 poi 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*, 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 `setpgrp`,<sup>6</sup> il cui prototipo è:

```
#include <unistd.h>
int setpgrp(void)
    Modifica il pgid al valore del pid del processo corrente.
```

La funzione restituisce il valore del nuovo *process group*.

La funzione, assegnando al *pgid* il valore del *pid* processo corrente, rende questo *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. È possibile invece spostare un processo da un raggruppamento ad un altro con la funzione `setpgid`, il cui prototipo è:

```
#include <unistd.h>
int setpgid(pid_t pid, pid_t pgid)
    Assegna al pgid del processo pid il valore pgid.
```

La funzione ritorna il valore del nuovo *process group*, e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

**ESRCH** Il processo selezionato non esiste.

**EPERM** Il cambiamento non è consentito.

**EACCES** Il processo ha già eseguito una `exec`.

**EINVAL** Il valore di *pgid* è negativo.

La funzione permette di cambiare il *pgid* del processo *pid*, ma il cambiamento può essere effettuato solo se *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`.<sup>7</sup> Specificando

<sup>5</sup>la system call è stata introdotta in Linux a partire dalla versione 1.3.44, il supporto nelle librerie del C è iniziato dalla versione 5.2.19. La funzione non è prevista da POSIX.1, che parla solo di processi leader di sessione, e non di identificatori di sessione.

<sup>6</sup>questa è la definizione di POSIX.1, BSD definisce una funzione con lo stesso nome, che però è identica a `setpgid`; nelle *glibc* viene sempre usata sempre questa definizione, a meno di non richiedere esplicitamente la compatibilità all'indietro con BSD, definendo la macro `_BSD_SOURCE`.

<sup>7</sup>questa caratteristica è implementata dal kernel che mantiene allo scopo un altro campo, `did_exec`, in `task_struct`.

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; pertanto `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.

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 di `setsid`; il suo prototipo è:

```
#include <unistd.h>
pid_t setsid(void)
```

Crea una nuova sessione sul processo corrente impostandone `sid` e `pgid`.

La funzione ritorna il valore del nuovo `sid`, e -1 in caso di errore, il solo errore possibile è `EPERM`, che si ha quando 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<sup>8</sup> se il suo `sid` è uguale al suo `pid`) ed unico componente. Inoltre la funzione distacca il processo da ogni terminale di controllo (torneremo sull'argomento in sez. 10.1.3) 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.<sup>9</sup> Questa funzione viene usata di solito nel processo di login (per i dettagli vedi sez. 10.1.4) per raggruppare in una sessione tutti i comandi eseguiti da un utente dalla sua shell.

### 10.1.3 Il terminale di controllo e il controllo di sessione

Come accennato in sez. 10.1.1, 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,<sup>10</sup> e dal quale ricevono gli eventuali segnali da tastiera.

A tale scopo 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.<sup>11</sup> In generale ogni processo eredita dal padre, insieme al `pgid` e al `sid` anche il terminale di controllo (vedi sez. 3.2.2). 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

<sup>8</sup>in Linux la proprietà è mantenuta in maniera indipendente con un apposito campo `leader` in `task_struct`.

<sup>9</sup>potrebbe sorgere il dubbio che, per il riutilizzo dei valori dei `pid` fatto nella creazione dei nuovi processi (vedi sez. 3.2.1), 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.

<sup>10</sup>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.

<sup>11</sup>Lo standard POSIX.1 non specifica nulla riguardo l'implementazione; in Linux anch'esso viene mantenuto nella solita struttura `task_struct`, nel campo `tty`.

riottenere<sup>12</sup>, un terminale di controllo. In generale questo viene fatto automaticamente dal sistema<sup>13</sup> quando viene aperto il primo terminale (cioè uno dei vari file di dispositivo `/dev/tty*`) che diventa automaticamente il terminale di controllo, mentre il processo diventa il *processo di controllo* di quella sessione.

In genere (a meno di redirezioni) nelle sessioni di lavoro questo terminale è associato ai file standard (di 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:

```
#include <unistd.h>
#include <termios.h>
int tcsetpgrp(int fd, pid_t pgrp_id)
    Imposta a pgrp_id il process group di foreground del terminale associato al file descriptor fd.
```

La funzione restituisce 0 in caso di successo, e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

- `ENOTTY` Il file `fd` non corrisponde al terminale di controllo del processo chiamante.
  - `ENOSYS` Il sistema non supporta il job control.
  - `EPERM` Il *process group* specificato non è nella stessa sessione del processo chiamante.
- ed inoltre `EBADF` ed `EINVAL`.

la funzione può essere eseguita con successo solo da un processo nella stessa sessione e con lo stesso terminale di controllo.

Come accennato in sez. 10.1.1, 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. 9.2.6) è di fermare il processo, di norma questo comporta che tutti i membri del gruppo verranno fermati, ma non si avranno condizioni di errore.<sup>14</sup> Se però si bloccano o ignorano i due segnali citati, le funzioni di lettura e scrittura falliranno con un errore di `EIO`.

Un processo può controllare qual'è il gruppo di *foreground* associato ad un terminale con la funzione `tcgetpgrp`, il cui prototipo è:

```
#include <unistd.h>
#include <termios.h>
pid_t tcgetpgrp(int fd)
    Legge il process group di foreground del terminale associato al file descriptor fd.
```

La funzione restituisce in caso di successo il `pgid` del gruppo di *foreground*, e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

- `ENOTTY` Non c'è un terminale di controllo o `fd` non corrisponde al terminale di controllo del processo chiamante.
- ed inoltre `EBADF` ed `ENOSYS`.

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

<sup>12</sup>solo quando ciò è necessario, cosa che, come vedremo in sez. 10.1.5, non è sempre vera.

<sup>13</sup>a meno di non avere richiesto esplicitamente che questo non diventi un terminale di controllo con il flag `O_NOCTTY` (vedi sez. 6.2.1). In questo Linux segue la semantica di SVr4; BSD invece richiede che il terminale venga allocato esplicitamente con una `ioctl` con il comando `TIOCSCTTY`.

<sup>14</sup>la shell in genere notifica comunque un avvertimento, avvertendo la presenza di processi bloccati grazie all'uso di `waitpid`.

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 redirezione 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. 9.2.6) 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 han 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.<sup>15</sup>

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. 3.2.4, tutti i suoi figli vengono adottati da `init`, che è fuori dalla sessione. Questo renderà

---

<sup>15</sup>l'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. 3.2.4.

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.

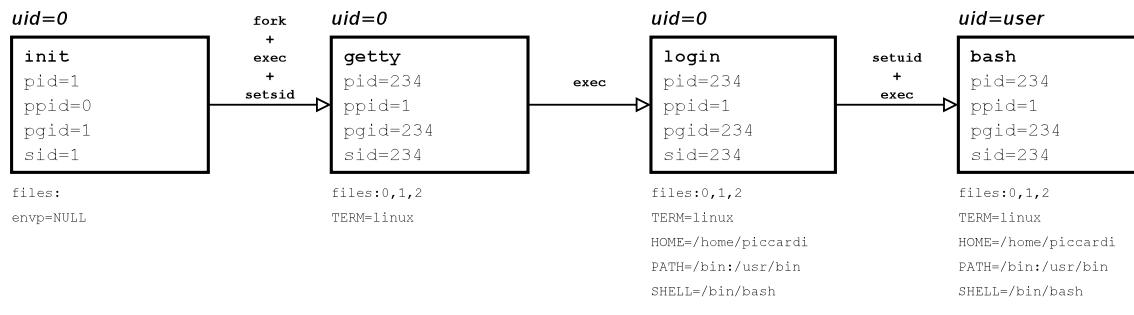
#### 10.1.4 Dal login alla shell

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<sup>16</sup> nel dispositivo cui il kernel associa i file standard (vedi sez. 6.1.2) per l'I/O, tratteremo solo il caso classico del terminale.

Abbiamo già brevemente illustrato in sez. 1.1.3 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<sup>17</sup> viene usata la procedura di avvio di System V; 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`) 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. 10.1.



**Figura 10.1:** 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, viene sempre visto attraverso un device driver che ne presenta un'interfaccia comune su un apposito file di dispositivo.

Per controllare un terminale si usa di solito il programma `getty` (od una delle sue varianti), che permette di mettersi in ascolto su uno di questi dispositivi. 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 su un terminale, il tutto ripetuto per ciascuno

<sup>16</sup>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.

<sup>17</sup>fa eccezione la distribuzione *Slackware*, come alcune distribuzioni su dischetto, ed altre distribuzioni dedicate a compiti limitati e specifici.

dei terminali che si hanno a disposizione (o per un certo numero di essi, nel caso delle console virtuali), 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 settaggi.<sup>18</sup> 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 `execve`, 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`) ad 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,<sup>19</sup> e richiede una password. Se l'utente non esiste o se la password non corrisponde<sup>20</sup> 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 settare 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. 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. 3.3.2, avendo invocato tali funzioni con i privilegi di amministratore, tutti gli user-ID ed i group-ID (reali, effettivi e salvati) saranno settati 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 sez. 6.1.2 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. 10.1.1.

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.

### 10.1.5 Prescrizioni per un programma *daemon*

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

---

<sup>18</sup>ad 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).

<sup>19</sup>in genere viene chiamata `getpwnam`, che abbiamo visto in sez. 8.2.3, per leggere la password e gli altri dati dal database degli utenti.

<sup>20</sup>il 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.

omonimi spiritelli che svolgevano compiti vari, di cui parlava Socrate (che sosteneva di averne uno al suo servizio).<sup>21</sup>

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 appena visto in sez. 10.1.3 (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 prescrizioni<sup>22</sup> 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`, introdotta per la prima volta in BSD4.4; il suo prototipo è:

```
#include <unistd.h>
int daemon(int nochdir, int noclose)
    Esegue le operazioni che distaccano il processo dal terminale di controllo e lo fanno girare
    come demone.

La funzione restituisce (nel nuovo processo) 0 in caso di successo, e -1 in caso di errore, nel qual
caso errno assumerà 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

<sup>21</sup>NdT. ricontrillare, i miei ricordi di filosofia sono piuttosto datati.

<sup>22</sup>ad esempio sia Stevens in [1], che la *Unix Programming FAQ* [9] ne riportano di sostanzialmente identiche.

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 permettesse 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, `syslogd`, che è anch'esso un *demone*. In generale i messaggi di errore vengono raccolti dal file speciale `/dev/log`, un *socket* locale (vedi sez. 14.3.4) dedicato a questo scopo, o via rete, con un *socket* UDP, o da un apposito demone, `klogd`, che estrae i messaggi del kernel.<sup>23</sup>

Il servizio permette poi di trattare i vari messaggi classificandoli attraverso due indici; il primo, chiamato *facility*, suddivide in diverse categorie i vari demoni in modo di raggruppare i messaggi 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 di *syslog* attraverso `syslogd` provvede poi a riportare i messaggi all'amministratore attraverso una serie differenti meccanismi come:

- scrivere sulla console.
- inviare via mail ad uno specifico utente.
- scrivere su un file (comunemente detto *log file*).
- inviare ad un altro demone (anche via rete).
- scartare.

secondo le modalità che questo preferisce e che possono essere impostate attraverso il file di configurazione `/etc/syslog.conf` (maggiori dettagli si possono trovare sulle pagine di manuale per questo file e per `syslogd`).

Le *glibc* definiscono 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 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 apre 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)
    Apre una connessione al sistema di syslog.

La funzione non restituisce nulla.
```

---

<sup>23</sup>i messaggi del kernel sono tenuti in un buffer circolare e scritti tramite la funzione `printk`, analoga alla `printf` usata in user space; una trattazione eccellente dell'argomento si trova in [4], nel quarto capitolo.

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]`); 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 impredicibili, 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. 10.1, il valore di `facility` deve essere specificato con un OR aritmetico.

Valore	Significato
<code>LOG_AUTH</code>	Messaggi relativi ad autenticazione e sicurezza, obsoleto, è sostituito da <code>LOG_AUTHPRIV</code> .
<code>LOG_AUTHPRIV</code>	Sostituisce <code>LOG_AUTH</code> .
<code>LOG_CRON</code>	Messaggi dei demoni di gestione dei comandi programmati ( <code>cron</code> e <code>at</code> ).
<code>LOG_DAEMON</code>	Demoni di sistema.
<code>LOG_FTP</code>	Server FTP.
<code>LOG_KERN</code>	Messaggi del kernel
<code>LOG_LOCAL0</code>	Riservato all'amministratore per uso locale
—	
<code>LOG_LOCAL7</code>	Riservato all'amministratore per uso locale
<code>LOG_LPR</code>	Messaggi del sistema di gestione delle stampanti
<code>LOG_MAIL</code>	Messaggi del sistema di posta elettronica
<code>LOG_NEWS</code>	Messaggi del sistema di gestione delle news (USENET)
<code>LOG_SYSLOG</code>	Messaggi generati dallo stesso <code>syslogd</code>
<code>LOG_USER</code>	Messaggi generici a livello utente
<code>LOG_UUCP</code>	Messaggi del sistema UUCP

**Tabella 10.1:** 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. 10.2.

Valore	Significato
<code>LOG_CONS</code>	Scrive sulla console quando.
<code>LOG_NDELAY</code>	Sostituisce <code>LOG_AUTH</code> .
<code>LOG_NOWAIT</code>	Messaggi dei demoni di gestione dei comandi programmati ( <code>cron</code> e <code>at</code> ).
<code>LOG_ODELAY</code>	.
<code>LOG_PERROR</code>	Stampa anche su <code>stderr</code> .
<code>LOG_PID</code>	Inserisce nei messaggi il <code>pid</code> del processo chiamante.

**Tabella 10.2:** Valori possibili per l'argomento `option` di `openlog`.

La funzione che si usa per generare un messaggio è `syslog`, dato che l'uso di `openlog` è opzionale, sarà quest'ultima a provvedere a chiamare la prima qualora ciò non sia stato fatto (nel qual caso il valore di `ident` è nullo). Il suo prototipo è:

```
#include <syslog.h>
void syslog(int priority, const char *format, ...)
```

Genera un messaggio di priorità `priority`.

La funzione non restituisce nulla.

Il comportamento della funzione è analogo 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. 7.2.6); 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`.

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<sup>24</sup> specificabile attraverso le costanti riportate in tab. 10.3. 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. 10.1.

Valore	Significato
<code>LOG_EMERG</code>	Il sistema è inutilizzabile.
<code>LOG_ALERT</code>	C'è una emergenza che richiede intervento immediato.
<code>LOG_CRIT</code>	Si è in una condizione critica.
<code>LOG_ERR</code>	Si è in una condizione di errore.
<code>LOG_WARNING</code>	Messaggio di avvertimento.
<code>LOG_NOTICE</code>	Notizia significativa relativa al comportamento.
<code>LOG_INFO</code>	Messaggio informativo.
<code>LOG_DEBUG</code>	Messaggio di debug.

**Tabella 10.3:** Valori possibili per l'indice di importanza del messaggio da specificare nell'argomento `priority` di `syslog`.

Una ulteriore funzione, `setlogmask`, permette di filtrare preliminarmente i messaggi in base alla loro priorità; il suo prototipo è:

```
#include <syslog.h>
int setlogmask(int mask)
    Imposta la maschera dei log al valore specificato.

La funzione restituisce il precedente valore.
```

Le routine di gestione mantengono per ogni processo una maschera che determina quale delle chiamate effettuate a `syslog` verrà effettivamente registrata. La registrazione viene disabilitata per tutte quelle priorità che non rientrano nella maschera; questa viene settata usando la macro `LOG_MASK(p)` dove `p` è una delle costanti di tab. 10.3. È inoltre disponibile anche la macro `LOG_UPTO(p)` che permette di specificare automaticamente tutte le priorità fino ad un certo valore.

## 10.2 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.

<sup>24</sup>le *glibc*, seguendo POSIX.1-2001, prevedono 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*.

### 10.2.1 L'architettura

I terminali sono una classe speciale di dispositivi a caratteri (si ricordi la classificazione di sez. 4.1.2); un terminale ha infatti una caratteristica che lo contraddistingue da un qualunque altro dispositivo, e cioè che è destinato a gestire l'interazione con un utente (deve essere cioè in grado di fare da terminale di controllo per una sessione), che comporta la presenza di ulteriori capacità.

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*`.<sup>25</sup> 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, 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 comportano 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;<sup>26</sup> ed in cui alcuni caratteri vengono interpretati per compiere operazioni (come la generazione dei segnali illustrati in sez. 9.2.6), questa di norma è la modalità in cui funziona 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 dai programmi (gli editor ad esempio) che necessitano di poter leggere un carattere alla volta e che gestiscono al loro interno i vari comandi.

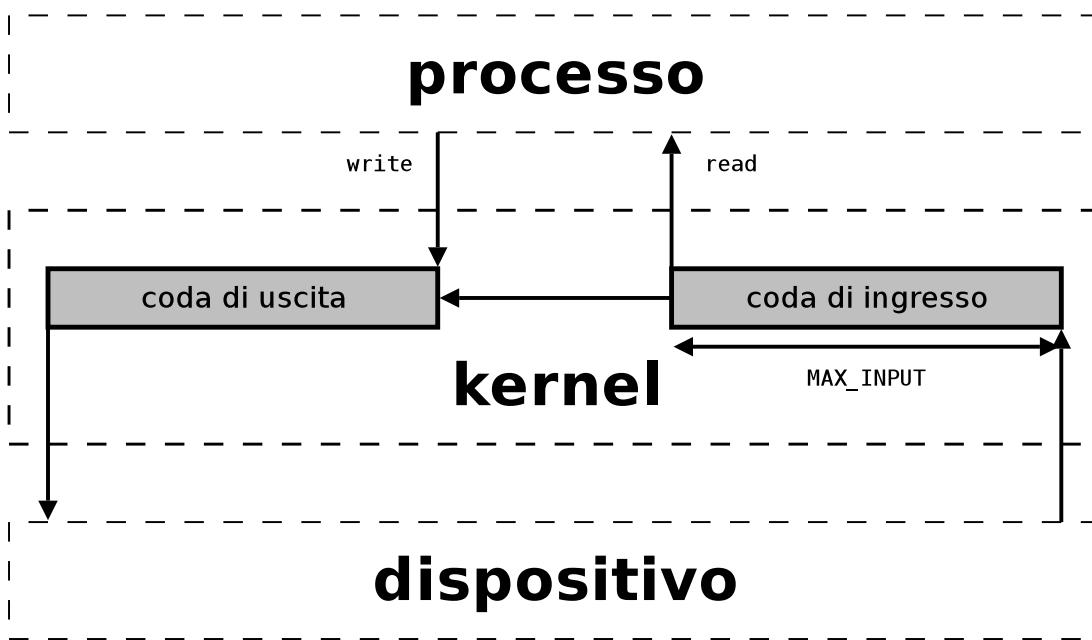
Per capire le caratteristiche dell'I/O sui terminali, occorre esaminare le modalità con cui esso viene effettuato; l'accesso, come per tutti i dispositivi, viene gestito da un driver apposito, la cui struttura generica è mostrata in fig. 10.2. Ad un terminale sono sempre associate due code per gestire l'input e l'output, che ne implementano una bufferizzazione<sup>27</sup> all'interno del kernel.

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. 8.1.3), 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 driver 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 driver 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 il modo canonico i caratteri in ingresso restano nella coda fintanto che non viene ricevuto un a capo; un'altra parametro del sistema, `MAX_CANON`, specifica la dimensione massima di una riga in modo canonico.

<sup>25</sup>ciò vale solo in parte per i terminali virtuali, essi infatti hanno due lati, un *master*, che può assumere i nomi `/dev/pty[p-zA-f]` [0-9a-f] ed un corrispondente *slave* con nome `/dev/tty[p-zA-f]` [0-9a-f].

<sup>26</sup>per cui 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 completa e la funzione ritornerà.

<sup>27</sup>completamente indipendente dalla eventuale ulteriore bufferizzazione fornita dall'interfaccia standard dei file.



*Figura 10.2:* Struttura interna generica di un driver per un terminale.

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 driver 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.

### 10.2.2 La gestione delle caratteristiche di un terminale

Data le loro peculiarità, fin dall'inizio si è posto il problema di come gestire le caratteristiche specifiche dei terminali; storicamente i vari dialetti di Unix hanno utilizzato diverse funzioni, alla fine con POSIX.1, è stata effettuata una standardizzazione, unificando le differenze fra BSD e System V in una unica interfaccia, che è quella usata dal Linux.

Alcune di queste funzioni prendono come argomento un file descriptor (in origine molte operazioni venivano effettuate con `ioctl`), ma ovviamente possono essere usate solo con file che corrispondano effettivamente ad un terminale (altrimenti si otterrà un errore di `ENOTTY`); questo può essere evitato utilizzando la funzione `isatty`, il cui prototipo è:

```
#include <unistd.h>
int isatty(int desc)
    Controlla se il file descriptor desc è un terminale.

La funzione restituisce 1 se desc è connesso ad un terminale, 0 altrimenti.
```

Un'altra funzione che fornisce informazioni su un terminale è `ttynname`, che permette di ottenere il nome del terminale associato ad un file descriptor; il suo prototipo è:

```
#include <unistd.h>
char *ttynname(int desc)
    Restituisce il nome del terminale associato al file desc.

La funzione restituisce il puntatore alla stringa contenente il nome del terminale associato desc e NULL in caso di errore.
```

Si tenga presente che la funzione restituisce un indirizzo di dati statici, che pertanto possono

essere sovrascritti da successive chiamate. Una funzione funzione analoga, anch'essa prevista da POSIX.1, è `ctermid`, il cui prototipo è:

```
#include <stdio.h>
char *ctermid(char *s)
    Restituisce il nome del terminale di controllo del processo.

La funzione restituisce il puntatore alla stringa contenente il pathname del terminale.
```

La funzione scrive il *pathname* del terminale di controllo del processo chiamante nella stringa posta all'indirizzo specificato dall'argomento `s`. La memoria per contenere la stringa deve essere stata allocata in precedenza ed essere lunga almeno `L_ctermid`<sup>28</sup> caratteri.

Esiste infine una versione rientrante `ttynname_r` della funzione `ttynname`, che non presenta il problema dell'uso di una zona di memoria statica; il suo prototipo è:

```
#include <unistd.h>
int ttynname_r(int desc, char *buff, size_t len)
    Restituisce il nome del terminale associato al file desc.

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso errno assumerà i valori:
ERANGE    la lunghezza del buffer, len, non è sufficiente per contenere la stringa restituita.
ed inoltre EBADF ed ENOSYS.
```

La funzione prende due argomenti, il puntatore alla zona di memoria `buff`, in cui l'utente vuole che il risultato venga scritto (dovrà ovviamente essere stata allocata in precedenza), e la relativa dimensione, `len`; se la stringa che deve essere restituita eccede questa dimensione si avrà una condizione di errore.

Se si passa come argomento `NULL` la funzione restituisce il puntatore ad una stringa statica che può essere sovrascritta da chiamate successive. Si tenga presente che il *pathname* restituito potrebbe non identificare univocamente il terminale (ad esempio potrebbe essere `/dev/tty`), inoltre non è detto che il processo possa effettivamente aprire il terminale.

I vari attributi vengono mantenuti per ciascun terminale in una struttura `termios`, (la cui definizione è riportata in fig. 10.3), usata dalle varie funzioni dell'interfaccia. In fig. 10.3 si sono riportati tutti i campi della definizione 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.<sup>29</sup>

---

```
struct termios {
    tcflag_t c_iflag;          /* input modes */
    tcflag_t c_oflag;          /* output modes */
    tcflag_t c_cflag;          /* control modes */
    tcflag_t c_lflag;          /* local modes */
    cc_t c_cc[NCCS];           /* control characters */
    cc_t c_line;               /* line discipline */
    speed_t c_ispeed;          /* input speed */
    speed_t c_ospeed;          /* output speed */
};
```

---

*Figura 10.3:* La struttura `termios`, che identifica le proprietà di un terminale.

<sup>28</sup>`L_ctermid` è una delle varie costanti del sistema, non trattata esplicitamente in sez. 8.1 che indica la dimensione che deve avere una stringa per poter contenere il nome di un terminale.

<sup>29</sup>la 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 ... (NdT, trovare a che serve).

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.

Valore	Significato
<code>INPCK</code>	Abilita il controllo di parità in ingresso. Se non viene impostato non viene fatto nessun controllo ed i caratteri vengono passati in input direttamente.
<code>IGNPAR</code>	Ignora gli errori di parità, il carattere viene passato come ricevuto. Ha senso solo se si è impostato <code>INPCK</code> .
<code>PARMRK</code>	Controlla come vengono riportati gli errori di parità. Ha senso solo se <code>INPCK</code> è impostato e <code>IGNPAR</code> no. Se impostato inserisce una sequenza <code>0xFF 0x00</code> prima di ogni carattere che presenta errori di parità, se non impostato un carattere con errori di parità viene letto come uno <code>0x00</code> . Se un carattere ha il valore <code>0xFF</code> e <code>ISTRIP</code> non è settato, per evitare ambiguità esso viene sempre riportato come <code>0xFF 0xFF</code> .
<code>ISTRIP</code>	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.
<code>IGNBRK</code>	Ignora le condizioni di <code>BREAK</code> 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.
<code>BRKINT</code>	Controlla la reazione ad un <code>BREAK</code> quando <code>IGNBRK</code> non è impostato. Se <code>BRKINT</code> è impostato il <code>BREAK</code> causa lo scarico delle code, e se il terminale è il terminale di controllo per un gruppo in foreground anche l'invio di <code>SIGINT</code> ai processi di quest'ultimo. Se invece <code>BRKINT</code> non è impostato un <code>BREAK</code> viene letto come un carattere NUL, a meno che non sia settato <code>PARMRK</code> nel qual caso viene letto come la sequenza di caratteri <code>0xFF 0x00 0x00</code> .
<code>IGNCR</code>	Se impostato il carattere di ritorno carrello ( <i>carriage return</i> , <code>'\r'</code> ) viene scartato dall'input. Può essere utile per i terminali che inviano entrambi i caratteri di ritorno carrello e a capo ( <i>newline</i> , <code>'\n'</code> ).
<code>ICRNL</code>	Se impostato un carattere di ritorno carrello ( <code>'\r'</code> ) sul terminale viene automaticamente trasformato in un a capo ( <code>'\n'</code> ) sulla coda di input.
<code>INLCR</code>	Se impostato il carattere di a capo ( <code>'\n'</code> ) viene automaticamente trasformato in un ritorno carrello ( <code>'\r'</code> ).
<code>IUCLC</code>	Se impostato trasforma i caratteri maiuscoli dal terminale in minuscoli sull'ingresso (opzione non POSIX).
<code>IXON</code>	Se impostato attiva il controllo di flusso in uscita con i caratteri di <code>START</code> e <code>STOP</code> . se si riceve uno <code>STOP</code> l'output viene bloccato, e viene fatto ripartire solo da uno <code>START</code> , 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.
<code>IXANY</code>	Se impostato con il controllo di flusso permette a qualunque carattere di far ripartire l'output bloccato da un carattere di <code>STOP</code> .
<code>IXOFF</code>	Se impostato abilita il controllo di flusso in ingresso. Il computer emette un carattere di <code>STOP</code> per bloccare l'input dal terminale e lo sblocca con il carattere <code>START</code> .
<code>IMAXBEL</code>	Se impostato fa suonare il cicalino se si riempie la cosa di ingresso; in Linux non è implementato e il kernel si comporta cose se fosse sempre settato (è una estensione BSD).

**Tabella 10.4:** Costanti identificative dei vari bit del flag di controllo `c_iflag` delle modalità di input di un terminale.

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. 10.4.

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.

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).
OCRNL	Se impostato converte automaticamente il carattere di a capo (NL) nella coppia di caratteri ritorno carrello, a capo (CR-NL).
OLCUC	Se impostato trasforma i caratteri minuscoli in ingresso in caratteri maiuscoli sull'uscita (non previsto da POSIX.1).
ONLCR	Se impostato converte automaticamente il carattere di a capo (NL) in un carattere di ritorno carrello (CR).
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).
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 10.5:** Costanti identificative dei vari bit del flag di controllo `c_oflag` delle modalità di output 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. 10.5.

Si noti come alcuni dei valori riportati in tab. 10.5 fanno riferimento a delle maschere di bit; essi infatti vengono utilizzati per impostare alcuni valori numerici relativi ai ritardi 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 setta 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, i settaggi 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. 10.6.

Valore	Significato
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 <code>open</code> senza aver specificato il flag di <code>O_NOBLOCK</code> si bloccherà il processo finché non si è stabilita una connessione con il modem; inoltre se viene rilevata una disconnessione viene inviato un <code>SIGHUP</code> 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 <code>EIO</code> .
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.
CREAD	Se è impostato si può leggere l'input del terminale, altrimenti i caratteri in ingresso vengono scartati quando arrivano.
CSTOPB	Se impostato vengono usati due bit di stop sulla linea seriale, se non impostato ne viene usato soltanto uno.
PARENB	Se impostato abilita la generazione il controllo di parità. La reazione in caso di errori dipende dai relativi valori per <code>c_iflag</code> , riportati in tab. 10.4. 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 <code>PARENB</code> . Se impostato viene usata una parità è dispari, altrimenti viene usata una parità pari.
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.
CBAUD	Maschera dei bit (4+1) usati per impostare della velocità della linea (il <i>baud rate</i> ) in ingresso. In Linux non è implementato in quanto viene usato un apposito campo di <code>termios</code> .
CBAUDEX	Bit aggiuntivo per l'impostazione della velocità della linea, per le stesse motivazioni del precedente non è implementato in Linux.
CIBAUD	Maschera dei bit della velocità della linea in ingresso. Analogico a <code>CBAUD</code> , anch'esso in Linux è mantenuto in un apposito campo di <code>termios</code> .
CRTSCTS	Abilita il controllo di flusso hardware sulla seriale, attraverso l'utilizzo delle dei due fili di RTS e CTS.

**Tabella 10.6:** Costanti identificative dei vari bit del flag di controllo `c_cflag` delle modalità di controllo di un terminale.

I valori di questo flag sono molto specifici, e completamente indirizzati al controllo di un terminale mantenuto su una linea seriale; essi pertanto non hanno nessuna rilevanza per i terminali che usano un'altra interfaccia, come le console virtuali e gli pseudo-terminali usati dalle connessioni di rete.

Inoltre alcuni valori 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. 10.3).

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. 10.7. 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 riconoscimento 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

Valore	Significato
ICANON	Se impostato il terminale opera in modo canonico, altrimenti opera in modo non canonico.
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).
ECHOPRT	Se impostato abilita la visualizzazione del carattere 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 verrà stampata una /.
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.
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.
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 prende 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).
ISIG	Se impostato abilita il riconoscimento dei caratteri INTR, QUIT, e SUSP generando il relativo segnale.
IEXTEN	Abilita alcune estensioni previste dalla implementazione. Deve essere impostato perché caratteri speciali come EOL2, LNEXT, REPRINT e WERASE possano essere interpretati.
NOFLSH	Se impostato disabilita lo scarico delle code di ingresso e uscita quando vengono emessi i segnali SIGINT, SIGQUIT and 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.
XCASE	Se settato 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.
DEFECCHO	Se impostate effettua l'eco solo se c'è un processo in lettura.
FLUSHO	Effettua la cancellazione della coda di uscita. Viene attivato dal carattere DISCARD. Non è supportato in Linux.
PENDIN	Indica che la linea deve essere ristampata, viene attivato dal carattere REPRINT e resta attivo fino alla fine della ristampa. Non è supportato in Linux.

**Tabella 10.7:** Costanti identificative dei vari bit del flag di controllo `c_lflag` delle modalità locali di un terminale.

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.<sup>30</sup>

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. 10.8, usando quelle definizioni diventa possibile assegnare un nuovo carattere di controllo con un codice del tipo:

```
value.c_cc[VEOL2] = '\n';
```

<sup>30</sup>in 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 <b>SIGINT</b> .
VQUIT	0x1C	(C-\)	Carattere di uscita provoca l'emissione di <b>SIGQUIT</b> .
VERASE	0x7f	DEL	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 <b>read</b> con zero caratteri, cioè la condizione di <i>end-of-file</i> .
VTIME	—	—	Timeout, in decimi di secondo, per una lettura in modo non canonico.
VMIN	—	—	Numero minimo di caratteri per una lettura in modo non canonico.
VSWTC	0x00	NUL	Carattere di switch. Non supportato in Linux.
VSTART	0x21	(C-q)	Carattere di START. Riavvia un output bloccato da uno STOP.
VSTOP	0x23	(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 <b>SIGTSTP</b> .
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.
VREPRINT	0x12	(C-r)	Ristampa i caratteri non ancora letti.
VDISCARD	0x07	(C-o)	Non riconosciuto in Linux.
VWERASE	0x17	(C-w)	Cancellazione di una parola.
VLNEXT	0x16	(C-v)	Carattere di escape, serve a quotare il carattere successivo che non viene interpretato ma passato direttamente all'output.
VEOL2	0x00	NUL	Ulteriore carattere di fine riga. Ha lo stesso effetto di <b>VEOL</b> ma può essere un carattere diverso.

**Tabella 10.8:** Valori dei caratteri di controllo mantenuti nel campo **c\_cc** della struttura **termios**.

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 settato **ISIG**; **VSTART** e **VSTOP** richiedono sia settato **IXON**; **VLNEXT**, **VWERASE**, **VREPRINT** richiedono sia settato **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.
```

Entrambe le funzioni restituiscono 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà i valori:

<b>EINTR</b>	La funzione è stata interrotta.
ed inoltre <b>EBADF</b> , <b>ENOTTY</b> ed <b>EINVAL</b> .	

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 impo-

stazioni 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. 10.9; 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
<code>TCSANOW</code>	Esegue i cambiamenti in maniera immediata.
<code>TCSADRAIN</code>	I cambiamenti vengono eseguiti dopo aver atteso che tutto l'output presente sulle code è stato scritto.
<code>TCSAFLUSH</code>	È identico a <code>TCSADRAIN</code> , ma in più scarta tutti i dati presenti sulla coda di input.

**Tabella 10.9:** 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.

In fig. 10.4 e fig. 10.5 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 di entrambe le funzioni può essere trovato nel file `SetTermAttr.c` dei sorgenti allegati.

La funzione `SetTermAttr` provvede ad impostare il bit specificato dall'argomento `flag`; prima si leggono i valori correnti (10) con `tcgetattr`, uscendo con un messaggio in caso di errore (11-14), poi si provvede a impostare solo i bit richiesti (possono essere più di uno) con un OR binario (15); infine si scrive il nuovo valore modificato con `tcsetattr` (16), notificando un eventuale errore (11-14) o uscendo normalmente.

La seconda funzione, `UnSetTermAttr`, è assolutamente identica alla prima, solo che in questo caso, in (15), 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

---

```

1 #include <unistd.h>
2 #include <termios.h>
3 #include <errno.h>
4
5 int SetTermAttr(int fd, tcflag_t flag)
6 {
7     struct termios values;
8     int res;
9     res = tcgetattr(fd, &values);
10    if (res) {
11        perror("Cannot get attributes");
12        return res;
13    }
14    values.c_lflag |= flag;
15    res = tcsetattr(fd, TCSANOW, &values);
16    if (res) {
17        perror("Cannot set attributes");
18        return res;
19    }
20    return 0;
21 }
```

---

**Figura 10.4:** Codice della funzione `SetTermAttr` che permette di impostare uno dei flag di controllo locale del terminale.

---

```

1 int UnSetTermAttr(int fd, tcflag_t flag)
2 {
3     struct termios values;
4     int res;
5     res = tcgetattr(fd, &values);
6     if (res) {
7         perror("Cannot get attributes");
8         return res;
9     }
10    values.c_lflag &= (~flag);
11    res = tcsetattr(fd, TCSANOW, &values);
12    if (res) {
13        perror("Cannot set attributes");
14        return res;
15    }
16    return 0;
17 }
```

---

**Figura 10.5:** Codice della funzione `UnSetTermAttr` che permette di rimuovere uno dei flag di controllo locale del terminale.

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 `cfsetsospeed` 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.
```

Entrambe le funzioni restituiscono 0 in caso di successo e -1 in caso di 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 (le *glibc* prevedono che i valori siano indicati in bit per secondo), ma in generale altre versioni di librerie 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.

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.
```

Entrambe le funzioni restituiscono 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**.

### 10.2.3 La gestione della disciplina di linea.

Come illustrato dalla struttura riportata in fig. 10.2 tutti i terminali hanno un insieme di funzionalità comuni, che prevedono la presenza di code di ingresso ed uscita; in generale si fa riferimento ad esse con il nome di *discipline di linea*.

Lo standard POSIX prevede alcune funzioni che permettono di intervenire direttamente sulla gestione di quest'ultime 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. 10.1.3.<sup>31</sup>

Una prima funzione, che è efficace solo in caso di terminali seriali asincroni (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 inviando un flusso di bit nulli.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà i valori **EBADF** o **ENOTTY**.

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.<sup>32</sup>

Le altre funzioni previste da POSIX servono a controllare il comportamento dell'interazione fra le code associate al terminale e l'utente; la prima è **tcdrain**, il cui prototipo è:

```
#include <unistd.h>
#include <termios.h>
int tcdrain(int fd)
    Attende lo svuotamento della coda di output.
```

La funzione restituisce 0 in caso di successo e -1 in caso di 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 restituisce 0 in caso di successo e -1 in caso di 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. 10.10, 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.

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:

<sup>31</sup>con la stessa eccezione, già vista per **tcsetattr**, che quest'ultimo sia bloccato o ignorato dal processo chiamante.

<sup>32</sup>POSIX specifica il comportamento solo nel caso si sia impostato un valore nullo per **duration**; il comportamento negli altri casi può dipendere dalla implementazione.

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 10.10:** Possibili valori per l'argomento `queue` della funzione `tcflush`.

```
#include <unistd.h>
#include <termios.h>
int tcflow(int fd, int action)
    Sospende e riavvia il flusso dei dati sul terminale.
```

La funzione restituisce 0 in caso di successo e -1 in caso di 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. 10.11.

Valore	Azione
TCOFF	Sospende l'output.
TCON	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 10.11:** Possibili valori per l'argomento `action` della funzione `tcflow`.

#### 10.2.4 Operare in modo non canonico

Operare con un terminale in modo canonico è relativamente semplice; basta eseguire una lettura e la funzione ritornerà quando una il driver del 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 dell'input è di norma eseguita direttamente dal driver del terminale, 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 tocca invece al 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 per 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. 10.8, 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.

## 10.3 La gestione dei terminali virtuali

Da fare.

### 10.3.1 I terminali virtuali

Qui vanno spiegati i terminali virtuali, `/dev/pty` e compagnia.

### 10.3.2 La funzione openpty

Qui vanno le cose su `openpty` e compagnia.

# Capitolo 11

## La gestione avanzata dei file

In questo capitolo affronteremo le tematiche relative alla gestione avanzata dei file. In particolare tratteremo delle funzioni di input/output avanzato, che permettono una gestione più sofisticata dell'I/O su file, a partire da quelle che permettono di gestire l'accesso contemporaneo a più file, per concludere con la gestione dell'I/O mappato in memoria. Dedicheremo poi la fine del capitolo alle problematiche del *file locking*.

### 11.1 L'I/O multiplexing

Uno dei problemi che si presentano quando si deve operare contemporaneamente su molti file usando le funzioni illustrate in cap. 6 e cap. 7 è che si può essere bloccati nelle operazioni su un file mentre un altro potrebbe essere disponibile. L'*I/O multiplexing* nasce risposta a questo problema. In questa sezione forniremo una introduzione a questa problematica ed analizzeremo le varie funzioni usate per implementare questa modalità di I/O.

#### 11.1.1 La problematica dell'I/O multiplexing

Abbiamo visto in sez. 9.3.1, affrontando la suddivisione fra *fast* e *slow* system call, che in certi casi le funzioni di I/O possono bloccarsi indefinitamente.<sup>1</sup> Ad esempio le operazioni di lettura possono bloccarsi quando non ci sono dati disponibili sul descrittore su cui si sta operando.

Questo comportamento causa uno dei problemi più comuni che ci si trova ad affrontare nelle operazioni di I/O, che si verifica quando si deve operare con più file descriptor eseguendo funzioni che possono bloccarsi senza che sia possibile prevedere quando questo può avvenire (il caso più classico è quello di un server in attesa di dati in ingresso da vari client). Quello che può accadere è di restare bloccati nell'eseguire una operazione su un file descriptor che non è “pronto”, quando ce ne potrebbe essere un’altro disponibile. 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 della operazione bloccata dipende da quanto si otterrebbe dal file descriptor “disponibile”) si potrebbe addirittura arrivare ad un *deadlock*.

Abbiamo già accennato in sez. 6.2.1 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 input/output 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

---

<sup>1</sup>si ricordi però che questo può accadere solo per le pipe, i socket ed alcuni file di dispositivo; sui file normali le funzioni di lettura e scrittura ritornano sempre subito.

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.

Per superare questo problema è stato introdotto il concetto di *I/O multiplexing*, una nuova modalità di operazioni che consente di tenere sotto controllo più file descriptor in contemporanea, permettendo di bloccare un processo quando le operazioni volute non sono possibili, e di riprenderne l'esecuzione una volta che almeno una di quelle richieste sia disponibile, 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 in sez. 15.6.

### 11.1.2 Le funzioni select e pselect

Il primo kernel unix-like ad introdurre una interfaccia per l'*I/O multiplexing* è stato BSD,<sup>2</sup> con la funzione **select**, il cui prototipo è:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
           timeval *timeout)
    Attende che uno dei file descriptor degli insiemi specificati diventi attivo.
```

La funzione in caso di successo restituisce il numero di file descriptor (anche nullo) che sono attivi, e -1 in caso di errore, nel qual caso **errno** assumerà uno dei valori:

<b>EBADF</b>	Si è specificato un file descriptor sbagliato in uno degli insiemi.
<b>EINTR</b>	La funzione è stata interrotta da un segnale.
<b>EINVAL</b>	Si è specificato per <b>n</b> un valore negativo o un valore non valido per <b>timeout</b> . ed inoltre <b>ENOMEM</b> .

La funzione mette il processo in stato di *sleep* (vedi tab. 3.5) 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. 9.4.2) identifica un insieme di segnali. Per la manipolazione di questi *file descriptor set* si possono usare delle opportune macro di preprocessore:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
FD_ZERO(fd_set *set)
    Inizializza l'insieme (vuoto).
FD_SET(int fd, fd_set *set)
    Inserisce il file descriptor fd nell'insieme.
FD_CLR(int fd, fd_set *set)
    Rimuove il file descriptor fd nell'insieme.
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<sup>3</sup>,

<sup>2</sup>la funzione **select** è apparsa in BSD4.2 e standardizzata in BSD4.4, ma è stata portata su tutti i sistemi che supportano i *socket*, compreso le varianti di System V.

<sup>3</sup>ad esempio in Linux, fino alla serie 2.0.x, c'era un limite di 256 file per processo.

ma da quando, come nelle versioni più recenti del kernel, non c'è più un limite massimo, esso indica le dimensioni massime dei numeri usati nei *file descriptor set*.<sup>4</sup> 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.

La funzione richiede di specificare tre insiemi distinti di file descriptor; il primo, `readfds`, verrà osservato per rilevare la disponibilità di effettuare una lettura,<sup>5</sup> il secondo, `writefd`s, per verificare la possibilità effettuare una scrittura ed il terzo, `exceptfds`, per verificare l'esistenza di eccezioni (come i messaggi urgenti su un *socket*, vedi sez. 18.1.1).

Dato che in genere non si tengono mai sotto controllo fino a `FD_SETSIZE` file contemporaneamente la funzione richiede di specificare qual'è il numero massimo dei 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 `n`, che deve corrispondere al valore massimo aumentato di uno.<sup>6</sup> Infine l'argomento `timeout`, 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.

La funzione restituisce il numero di file descriptor pronti,<sup>7</sup> 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 si ha un timeout viene restituito un valore nullo e gli insiemi non vengono modificati. In caso di errore la funzione restituisce -1, ed i valori dei tre insiemi sono indefiniti e non si può fare nessun affidamento sul loro contenuto.

In Linux `select` modifica anche il valore di `timeout`, impostandolo al tempo restante in caso di interruzione prematura; questo è utile quando la funzione viene interrotta da un segnale, in tal caso infatti si ha un errore di `EINTR`, ed occorre rilanciare la funzione; in questo modo non è necessario ricalcolare tutte le volte il tempo rimanente.<sup>8</sup>

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 `n` un valore 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 anche 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

<sup>4</sup>il suo valore, secondo lo standard POSIX 1003.1-2001, è definito in `sys/select.h`, ed è pari a 1024.

<sup>5</sup>per 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*.

<sup>6</sup>i file descriptor infatti sono contati a partire da zero, ed il valore indica il numero di quelli da tenere sotto controllo; dimenticarsi di aumentare di uno il valore di `n` è un errore comune.

<sup>7</sup>questo è il comportamento previsto dallo standard, ma la standardizzazione della funzione è recente, ed esistono ancora alcune versioni di Unix che non si comportano in questo modo.

<sup>8</sup>questo può causare problemi di portabilità sia quando si trasporta codice scritto su Linux che legge questo valore, sia quando si usano programmi scritti per altri sistemi che non dispongono di questa caratteristica e ricalcolano `timeout` tutte le volte. In genere la caratteristica è disponibile nei sistemi che derivano da System V e non disponibile per quelli che derivano da BSD.

relative vengano dichiarate nell'header `sys/select.h`, che sostituisce i precedenti, ed inoltre aggiunge a `select` una nuova funzione `pselect`,<sup>9</sup> 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(SIG_BLOCK, &sigmask))
    Attende che uno dei file descriptor degli insiemi specificati diventi attivo.
```

La funzione in caso di successo restituisce il numero di file descriptor (anche nullo) che sono attivi, e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	Si è specificato un file descriptor sbagliato in uno degli insiemi.
<code>EINTR</code>	La funzione è stata interrotta da un segnale.
<code>EINVAL</code>	Si è specificato per <code>n</code> un valore negativo o un valore non valido per <code>timeout</code> .

ed inoltre `ENOMEM`.

La funzione è sostanzialmente identica a `select`, solo che usa una struttura `timespec` (vedi fig. 8.9) per indicare con maggiore precisione il timeout e non ne aggiorna il valore in caso di interruzione. Inoltre prende un argomento aggiuntivo `sigmask` che è il puntatore ad una maschera di segnali (si veda sez. 9.4.4). La maschera corrente viene sostituita da questa immediatamente prima di eseguire l'attesa, e ripristinata al ritorno della funzione.

L'uso di `sigmask` è stato introdotto allo scopo di prevenire possibili race condition quando ci si deve porre in attesa sia di un segnale che di dati. La tecnica classica è quella di utilizzare il gestore per impostare una variabile globale e controllare questa nel corpo principale del programma; abbiamo visto in sez. 9.4.1 come questo lasci spazio a 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,<sup>10</sup> ribloccandolo non appena essa ritorna, così che il precedente codice potrebbe essere riscritto nel seguente modo:

<sup>9</sup>il supporto per lo standard POSIX 1003.1-2001, ed l'header `sys/select.h`, compaiono in Linux a partire dalle `libc` 2.1. Le `libc` 4 e `libc` 5 non contengono questo header, le `libc` 2.0 contengono una definizione sbagliata di `psignal`, senza l'argomento `sigmask`, la definizione corretta è presente dalle `libc` 2.1-2.2.1 se si è definito `_GNU_SOURCE` e nelle `libc` 2.2.2-2.2.4 se si è definito `_XOPEN_SOURCE` con valore maggiore di 600.

<sup>10</sup>in Linux però non è presente la relativa system call, e la funzione è implementata nelle `libc` attraverso `select` (vedi `man select_tut`) per cui la possibilità di race condition permane; esiste però una soluzione, chiamata `self-`

```

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.

### 11.1.3 La funzione poll

Nello sviluppo di System V, invece di utilizzare l'interfaccia di `select`, che è una estensione tipica di BSD, è stata introdotta un'altra interfaccia, basata sulla funzione `poll`,<sup>11</sup> il cui prototipo è:

```
#include <sys/poll.h>
int poll(struct pollfd *ufds, unsigned int nfds, int timeout)
    La funzione attende un cambiamento di stato su un insieme di file descriptor.
```

La funzione restituisce il numero di file descriptor con attività in caso di successo, o 0 se c'è stato un timeout e -1 in caso di errore, ed in quest'ultimo caso `errno` assumerà uno dei valori:

<code>EBADF</code>	Si è specificato un file descriptor sbagliato in uno degli insiemi.
<code>EINTR</code>	La funzione è stata interrotta da un segnale.
<code>EINVAL</code>	Il valore di <code>nfds</code> eccede il limite <code>RLIMIT_NOFILE</code> .
ed inoltre <code>EFAULT</code> e <code>ENOMEM</code> .	

La funzione permette di tenere sotto controllo contemporaneamente `ndfs` 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*).

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. 11.1, 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`. 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.

---

*pipe trick*, che consiste nell'aprire una pipe (vedi sez. 12.1.1) ed usare `select` sul capo in lettura della stessa, e indicare l'arrivo di un segnale scrivendo sul capo in scrittura all'interno del manipolatore; in questo modo anche se il segnale va perso prima della chiamata di `select` questa lo riconoscerà comunque dalla presenza di dati sulla pipe.

<sup>11</sup>la 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.

---

```
struct pollfd {
    int fd;           /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

---

**Figura 11.1:** La struttura `pollfd`, utilizzata per specificare le modalità di controllo di un file descriptor alla funzione `poll`.

Le costanti che definiscono i valori relativi ai bit usati nelle maschere binarie dei campi `events` e `revents` sono riportati in tab. 11.1, insieme al loro significato. Le si sono suddivise in tre gruppi, nel primo gruppo si sono indicati i bit utilizzati per controllare l'attività in ingresso, nel secondo quelli per l'attività in uscita, mentre il terzo gruppo contiene dei valori che vengono utilizzati solo nel campo `revents` per notificare delle condizioni di errore.

Flag	Significato
<code>POLLIN</code>	È possibile la lettura.
<code>POLLRDNORM</code>	Sono disponibili in lettura dati normali.
<code>POLLRDBAND</code>	Sono disponibili in lettura dati prioritari.
<code>POLLPRI</code>	È possibile la lettura di dati urgenti.
<code>POLLOUT</code>	È possibile la scrittura immediata.
<code>POLLWRNORM</code>	È possibile la scrittura di dati normali.
<code>POLLWRBAND</code>	È possibile la scrittura di dati prioritari.
<code>POLLERR</code>	C'è una condizione di errore.
<code>POLLHUP</code>	Si è verificato un hung-up.
<code>POLLNVAL</code>	Il file descriptor non è aperto.
<code>POLLMSG</code>	Definito per compatibilità con SysV.

**Tabella 11.1:** 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 SysV che usa gli *stream*.<sup>12</sup> è da questi che derivano i nomi di alcune costanti, in quanto per essi sono definite tre classi di dati: *normali*, *prioritari* ed *urgenti*. In Linux la distinzione ha senso solo per i dati *out-of-band* dei socket (vedi sez. 18.1.1), 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. Si tenga conto comunque che le costanti relative ai diversi tipi di dati (come `POLLRDNORM` e `POLLRDBAND`) sono utilizzabili soltanto qualora si sia definita la macro `_XOPEN_SOURCE`.<sup>13</sup>

In caso di successo funzione 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 (nel qual caso vengono utilizzati i valori di tab. 11.1 esclusivi di `revents`). Un valore 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`.

## 11.2 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

<sup>12</sup>essi sono una interfaccia specifica di SysV non presente in Linux, e non hanno nulla a che fare con i file *stream* delle librerie standard del C.

<sup>13</sup>e ci si ricordi di farlo sempre in testa al file, definirla soltanto prima di includere `sys/poll.h` non è sufficiente.

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), per essere avvisato della possibilità di eseguire le operazioni di I/O volute.

### 11.2.1 Operazioni asincrone sui file

Abbiamo accennato in sez. 6.2.1 che è possibile, attraverso l'uso del flag `O_ASYNC`<sup>14</sup>, aprire un file in modalità asincrona, così come è possibile attivare in un secondo tempo questa modalità impostando questo flag attraverso l'uso di `fcntl` con il comando `F_SETFL` (vedi sez. 6.3.5).

In realtà in questo caso non si tratta di eseguire delle operazioni di lettura o scrittura del file in modo asincrono (tratteremo questo, che più propriamente è detto *I/O asincrono* in sez. 11.2.2), quanto di un meccanismo asincrono di notifica delle variazioni dello stato del file descriptor aperto in questo modo.

Quello che succede in questo caso è che il sistema genera un segnale (normalmente `SIGIO`, ma è possibile usarne altri con il comando `F_SETSIG` di `fcntl`) tutte le volte che diventa possibile leggere o scrivere dal file descriptor che si è posto in questa modalità. Si può inoltre selezionare, con il comando `F_SETOWN` di `fcntl`, quale processo (o gruppo di processi) riceverà il segnale. Se pertanto si effettuano le operazioni di I/O in risposta alla ricezione del segnale non ci sarà più la necessità di restare bloccati in attesa della disponibilità di accesso ai file; per questo motivo Stevens chiama questa modalità *signal driven I/O*.

In questo modo si può evitare l'uso delle funzioni `poll` o `select` che, quando vengono usate con un numero molto grande di file descriptor, non hanno buone prestazioni. In tal caso infatti la maggior parte del loro tempo di esecuzione è 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.

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. 9.1.4), 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 (si riveda quanto illustrato in sez. 9.4.3).

Per far questo però occorre utilizzare le funzionalità dei segnali real-time (vedi sez. 9.4.6) 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`<sup>15</sup> di `siginfo_t`, troverà nel campo `si_fd` il valore del file descriptor che ha generato il segnale.

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

---

<sup>14</sup>l'uso del flag di `O_ASYNC` e dei comandi `F_SETOWN` e `F_GETOWN` per `fcntl` è specifico di Linux e BSD.

<sup>15</sup>il valore resta `SI_SIGIO` qualunque sia il segnale che si è associato all'I/O asincrono, ed indica appunto che il segnale è stato generato a causa di attività nell'I/O asincrono.

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 ciclo quali sono i file diventati attivi.

### 11.2.2 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*. 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 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, 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. Al momento esiste una sola versione stabile di questa interfaccia, quella delle *glibc*, che è realizzata completamente in user space, ed accessibile linkando i programmi con la libreria *librt*. Nei kernel della nuova serie è stato anche introdotto (a partire dal 2.5.32) un nuovo layer per l'I/O asincrono.

Lo standard 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. 11.2. Nello stesso file è definita la macro *\_POSIX\_ASYNCNCHRONOUS\_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 */
    int aio_lio_opcode;       /* Operation to be performed. */
    int aio_reqprio;          /* Request priority offset. */
    volatile void *aio_buf;    /* Location of buffer. */
    size_t aio_nbytes;        /* Length of transfer. */
    struct sigevent aio_sigevent; /* Signal number and value. */
};
```

---

Figura 11.2: 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 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 eseguite 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.<sup>16</sup> La priorità viene impostata a partire da quella del processo chiamante (vedi sez. 3.4), 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.

---

```
struct sigevent
{
    sigval_t sigev_value;
    int sigev_signo;
    int sigev_notify;
    void (*sigev_notify_function)(sigval_t);
    pthread_attr_t *sigev_notify_attributes;
};
```

---

**Figura 11.3:** La struttura `sigevent`, usata per specificare le modalità di notifica degli eventi relativi alle operazioni di I/O asincrono.

Infine il campo `aio_sigevent` è una struttura di tipo `sigevent` che serve a specificare il modo in cui si vuole che venga effettuata la notifica del completamento delle operazioni richieste. La struttura è riportata in fig. 11.3; il campo `sigev_notify` è quello che indica le modalità della notifica, esso può assumere i tre valori:

- SIGEV\_NONE** Non viene inviata nessuna notifica.
- SIGEV\_SIGNAL** La notifica viene effettuata inviando al processo chiamante il segnale specificato da `sigev_signo`; se il gestore di questo è stato installato con `SA_SIGINFO` gli verrà restituito il valore di `sigev_value` (la cui definizione è in fig. 9.13) come valore del campo `si_value` di `siginfo_t`.
- SIGEV\_THREAD** La notifica viene effettuata creando un nuovo thread che esegue la funzione specificata da `sigev_notify_function` con argomento `sigev_value`, e con gli attributi specificati da `sigev_notify_attributes`.

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:

```
#include <aio.h>
int aio_read(struct aiocb *aiocbp)
    Richiede una lettura asincrona secondo quanto specificato con aiocbp.
int aio_write(struct aiocb *aiocbp)
    Richiede una scrittura asincrona secondo quanto specificato con aiocbp.
```

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

- |               |   |
|---------------|---|
| <b>EBADF</b>  | Si è specificato un file descriptor sbagliato.  |
| <b>ENOSYS</b> | La funzione non è implementata.   |
| <b>EINVAL</b> | Si è specificato un valore non valido per i campi <code>aio_offset</code> o <code>aio_reqprio</code> di <code>aiocbp</code> . |
| <b>EAGAIN</b> | La coda delle richieste è momentaneamente piena.  |

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

<sup>16</sup>in generale perché ciò sia possibile occorre che la piattaforma supporti questa caratteristica, questo viene indicato definendo le macro `_POSIX_PRIORITIZED_IO`, e `_POSIX_PRIORITY_SCHEDULING`.

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. 6.2.1), nel qual caso le scritture vengono effettuate comunque alla fine de 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 impredicibili, 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 delle operazioni di I/O associate a aiocbp.
```

La funzione restituisce 0 se le operazioni si sono concluse con successo, altrimenti restituisce il codice di errore relativo al loro fallimento.

Se l'operazione non si è ancora completata viene restituito l'errore di `EINPROGRESS`. 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` e `fsync`.

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)
    Recupera il valore dello stato di ritorno delle operazioni di I/O associate a aiocbp.

    La funzione restituisce lo stato di uscita dell'operazione eseguita.
```

La funzione deve essere chiamata una sola volta per ciascuna operazione asincrona, essa infatti fa sì che il sistema rilasci le risorse ad essa associate. È per questo motivo che occorre chiamare la funzione solo dopo che l'operazione cui `aiocbp` fa riferimento si è completata. Una chiamata precedente il completamento delle operazioni darebbe risultati indeterminati.

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`). È 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 per il file indicato da aiocbp.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, che può essere, con le stesse modalità di `aio_read`, `EAGAIN`, `EBADF` o `EINVAL`.

La funzione richiede la sincronizzazione delle operazioni di I/O, ritornando immediatamente. 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. 6.3.3).

Il successo della chiamata assicura la sincronizzazione delle operazioni fino allora richieste, 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.

In alcuni casi può essere necessario interrompere le operazioni (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 fildes, struct aiocb *aiocbp)
    Richiede la cancellazione delle operazioni sul file fildes specificate da aiocbp.
```

La funzione restituisce il risultato dell'operazione con un codice di positivo, e -1 in caso di errore, che avviene qualora si sia specificato un valore non valido di `fildes`, imposta `errno` al valore `EBADF`.

La funzione permette di cancellare una operazione specifica sul file `fildes`, 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 il suo codice di ritorno sarà -1, inoltre il meccanismo di notifica non verrà invocato. Se si specifica una operazione relativa ad un altro file descriptor il risultato è indeterminato.

In caso di successo, i possibili valori di ritorno per `aio_cancel` sono tre (anch'essi definiti in `aio.h`):

`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, per un massimo di timeout, il completamento di una delle operazioni specificate
    da list.
```

La funzione restituisce 0 se una (o più) operazioni sono state completate, e -1 in caso di errore nel qual caso `errno` assumerà uno dei valori:

`EAGAIN` Nessuna operazione è stata completata entro `timeout`.

`ENOSYS` La funzione non è implementata.

`EINTR` La funzione è stata interrotta da un segnale.

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 da `timout`, o fintanto che non arrivi un segnale.<sup>17</sup> 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 `NUL` per `timout` comporta l'assenza di timeout.

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 delle operazioni di I/O elencata da `list`, secondo la modalità `mode`.

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

<code>EAGAIN</code>	Nessuna operazione è stata completata entro <code>timeout</code> .
<code>EINVAL</code>	Si è passato un valore di <code>mode</code> non valido o un numero di operazioni <code>nent</code> maggiore di <code>AIO_LISTIO_MAX</code> .
<code>ENOSYS</code>	La funzione non è implementata.
<code>EINTR</code>	La funzione è stata interrotta da un segnale.

La funzione esegue la richiesta delle `nent` operazioni indicate dalla lista `list`; questa deve contenere gli indirizzi di altrettanti *control block*, opportunamente inizializzati; in particolare nel caso dovrà essere specificato il tipo di operazione tramite il campo `aio_lio_opcode`, che può prendere i tre valori:

`LIO_READ` si richiede una operazione di lettura.

`LIO_WRITE` si richiede una operazione di scrittura.

`LIO_NOP` non si effettua nessuna operazione.

l'ultimo valore viene usato quando si ha a che fare con un vettore di dimensione fissa, per poter specificare solo alcune operazioni, o quando si è dovuto cancellare delle operazioni e si deve ripetere la richiesta per quelle non completate.

L'argomento `mode` permette di stabilire il comportamento della funzione, se viene specificato il valore `LIO_WAIT` la funzione si blocca fino al completamento di tutte le operazioni richieste; se invece si specifica `LIO_NOWAIT` la funzione ritorna immediatamente dopo aver messo in coda tutte le richieste. In questo 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`.

### 11.3 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. 6.2. In questa sezione allora prenderemo in esame le interfacce per l'*I/O vettorizzato* e per l'*I/O mappato in memoria*.

---

<sup>17</sup>si tenga conto che questo segnale può anche essere quello utilizzato come meccanismo di notifica.

### 11.3.1 I/O vettorizzato

Un caso abbastanza comune è quello in cui ci si trova a 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, ci sono casi in cui si vuole poter contare sulla atomicità delle operazioni.

Per questo motivo BSD 4.2<sup>18</sup> ha introdotto due nuove system call, `readv` e `writev`, che permettono di effettuare con una sola chiamata una lettura o una scrittura su una serie di buffer (quello che viene chiamato *I/O vettorizzato*). I relativi prototipi sono:

```
#include <sys/uio.h>
int readv(int fd, const struct iovec *vector, int count)
    Esegue una lettura vettorizzata da fd nei count buffer specificati da vector.
int writev(int fd, const struct iovec *vector, int count)
    Esegue una scrittura vettorizzata da fd nei count buffer specificati da vector.
```

Le funzioni restituiscono il numero di byte letti o scritti in caso di successo, e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	si è specificato un file descriptor sbagliato.
<code>EINVAL</code>	si è specificato un valore non valido per uno degli argomenti (ad esempio <code>count</code> è maggiore di <code>MAX_IOVEC</code> ).
<code>EINTR</code>	la funzione è stata interrotta da un segnale prima di avere eseguito una qualunque lettura o scrittura.
<code>EAGAIN</code>	<code>fd</code> è stato aperto in modalità non bloccante e non ci sono dati in lettura.
<code>EOPNOTSUPP</code>	La coda delle richieste è momentaneamente piena.

ed inoltre `EISDIR`, `ENOMEM`, `EFAULT` (se non sono stato allocati correttamente i buffer specificati nei campi `iov_base`), più tutti gli ulteriori errori che potrebbero avere le usuali funzioni di lettura e scrittura eseguite su `fd`.

Entrambe le funzioni usano una struttura `iovec`, definita in fig. 11.4, che definisce dove i dati devono essere letti o scritti. Il primo campo, `iov_base`, contiene l'indirizzo del buffer ed il secondo, `iov_len`, la dimensione dello stesso.

---

```
struct iovec {
    __ptr_t iov_base;      /* Starting address */
    size_t iov_len;        /* Length in bytes */
};
```

---

*Figura 11.4:* La struttura `iovec`, usata dalle operazioni di I/O vettorizzato.

I buffer da utilizzare sono indicati attraverso l'argomento `vector` che è un vettore di strutture `iovec`, la cui lunghezza è specificata da `count`. Ciascuna struttura dovrà essere inizializzata per opportunamente per indicare i vari buffer da/verso i quali verrà eseguito il trasferimento dei dati. Essi verranno letti (o scritti) nell'ordine in cui li si sono specificati nel vettore `vector`.

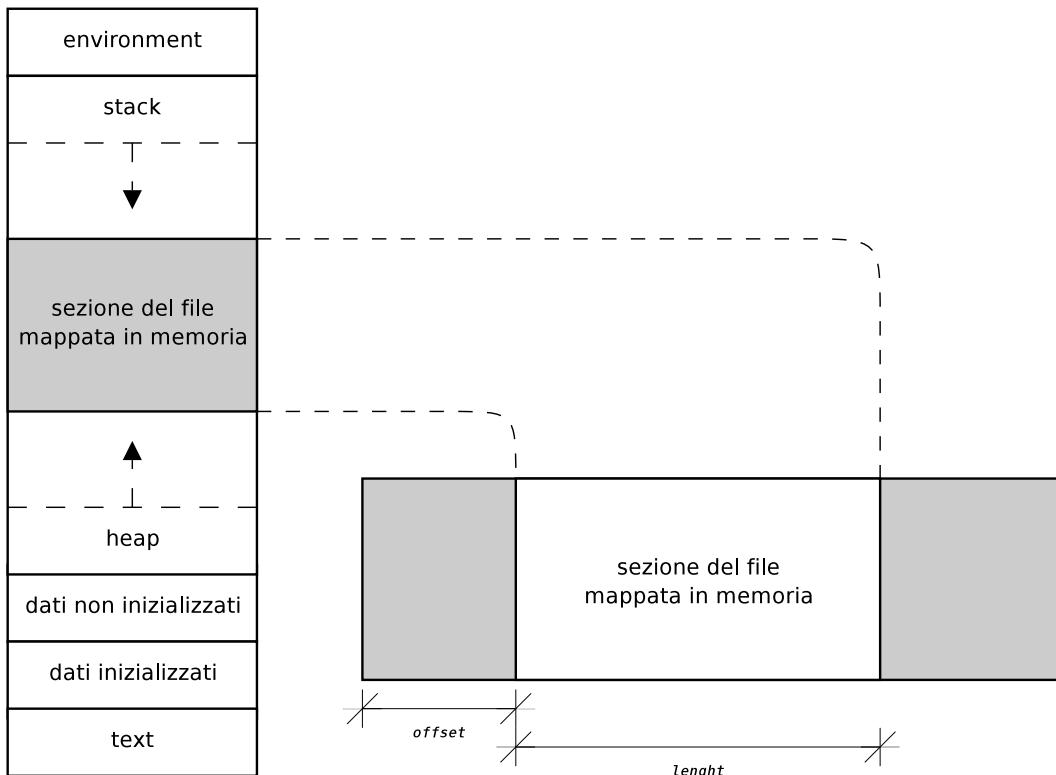
### 11.3.2 File mappati in memoria

Una modalità alternativa di I/O, che usa una interfaccia completamente diversa rispetto a quella classica vista in cap. 6, è il cosiddetto *memory-mapped I/O*, che, attraverso il meccanismo della

<sup>18</sup>Le due funzioni sono riprese da BSD4.4 ed integrate anche dallo standard Unix 98. Fino alle libc5, Linux usava `size_t` come tipo dell'argomento `count`, una scelta logica, che però è stata dismessa per restare aderenti allo standard.

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.

Il meccanismo è illustrato in fig. 11.5, 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 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*.



**Figura 11.5:** Disposizione della memoria di un processo quando si esegue la mappatura in memoria di un 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 per la gestione del *me-*

*mory 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 <unistd.h>
#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 del file fd.
```

La funzione restituisce il puntatore alla zona di memoria mappata in caso di successo, e `MAP_FAILED` (-1) in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EBADF</code>	Il file descriptor non è valido, e non si è usato <code>MAP_ANONYMOUS</code> .
<code>EACCES</code>	o <code>fd</code> non si riferisce ad un file regolare, o si è usato <code>MAP_PRIVATE</code> ma <code>fd</code> non è aperto in lettura, o si è usato <code>MAP_SHARED</code> e impostato <code>PROT_WRITE</code> ed <code>fd</code> non è aperto in lettura/scrittura, o si è impostato <code>PROT_WRITE</code> ed <code>fd</code> è in <i>append-only</i> .
<code>EINVAL</code>	I valori di <code>start</code> , <code>length</code> o <code>offset</code> non sono validi (o troppo grandi o non allineati sulla dimensione delle pagine).
<code>ETXTBSY</code>	Si è impostato <code>MAP_DENYWRITE</code> ma <code>fd</code> è aperto in scrittura.
<code>EAGAIN</code>	Il file è bloccato, o si è bloccata troppa memoria.
<code>ENOMEM</code>	Non c'è memoria o si è superato il limite sul numero di mappature possibili.
<code>ENODEV</code>	Il filesystem di <code>fd</code> non supporta il memory mapping.

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 di `offset` deve essere un multiplo della dimensione di una pagina di memoria.

Valore	Significato
<code>PROT_EXEC</code>	Le pagine possono essere eseguite.
<code>PROT_READ</code>	Le pagine possono essere lette.
<code>PROT_WRITE</code>	Le pagine possono essere scritte.
<code>PROT_NONE</code>	L'accesso alle pagine è vietato.

**Tabella 11.2:** Valori dell'argomento `prot` di `mmap`, relativi alla protezione applicate alle pagine del file mappate in memoria.

Il valore dell'argomento `prot` indica la protezione<sup>19</sup> 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. 11.3; 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 tab. 11.3.

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.

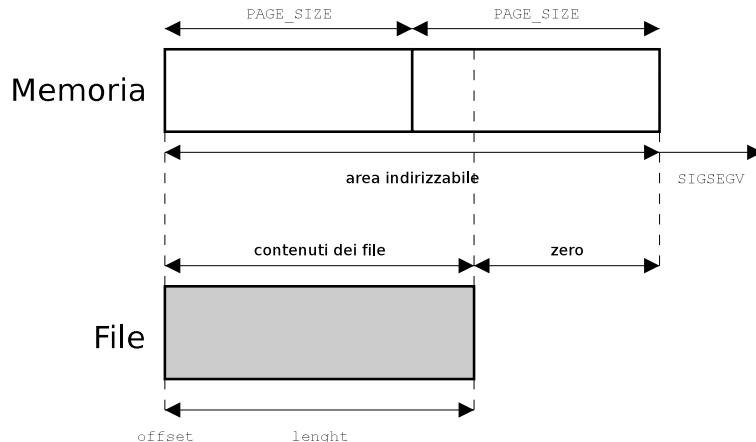
<sup>19</sup>In Linux la memoria reale è divisa in pagine: ogni processo vede la sua memoria attraverso uno o più segmenti lineari di memoria virtuale. Per ciascuno di questi segmenti il kernel mantiene nella *page table* la mappatura sulle pagine di memoria reale, ed le modalità di accesso (lettura, esecuzione, scrittura); una loro violazione causa quella che si chiama una *segment violation*, e la relativa emissione del segnale `SIGSEGV`.

<sup>21</sup>Dato che tutti faranno riferimento alle stesse pagine di memoria.

<sup>21</sup>L'uso di questo flag con `MAP_SHARED` è stato implementato in Linux a partire dai kernel della serie 2.4.x.

Valore	Significato
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_SHARED	I cambiamenti sulla memoria mappata vengono riportati sul file e saranno immediatamente visibili agli altri processi che mappano lo stesso file. <sup>20</sup> Il file su disco però non sarà aggiornato fino alla chiamata di <code>msync</code> o <code>unmap</code> ), e solo allora le modifiche saranno visibili per l'I/O convenzionale. Incompatibile con <code>MAP_PRIVATE</code> .
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. Le modifiche sono mantenute attraverso il meccanismo del <i>copy on write</i> e salvate su swap in caso di necessità. Non è specificato se i cambiamenti sul file originale vengano riportati sulla regione mappata. Incompatibile con <code>MAP_SHARED</code> .
MAP_DENYWRITE	In Linux viene ignorato per evitare <i>DoS</i> (veniva usato per segnalare che tentativi di scrittura sul file dovevano fallire con <code>ETXTBSY</code> ).
MAP_EXECUTABLE	Ignorato.
MAP_NORESERVE	Si usa con <code>MAP_PRIVATE</code> . Non riserva delle pagine di swap 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 <code>SIGSEGV</code> .
MAP_LOCKED	Se impostato impedisce lo swapping delle pagine mappate.
MAP_GROWSDOWN	Usato per gli stack. Indica che la mappatura deve essere effettuata con gli indirizzi crescenti verso il basso.
MAP_ANONYMOUS	La mappatura non è associata a nessun file. Gli argomenti <code>fd</code> e <code>offset</code> sono ignorati. <sup>21</sup>
MAP_ANON	Sinonimo di <code>MAP_ANONYMOUS</code> , deprecato.
MAP_FILE	Valore di compatibilità, deprecato.

**Tabella 11.3:** Valori possibili dell'argomento `flag` di `mmap`.



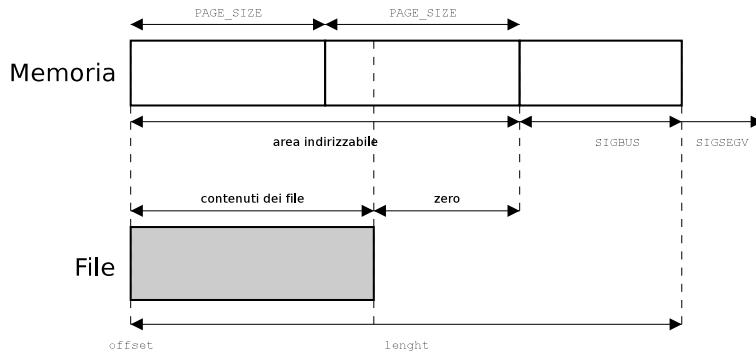
**Figura 11.6:** Schema della mappatura in memoria di una sezione di file di dimensioni non corrispondenti al bordo di una pagina.

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. Il caso più comune è quello illustrato in fig. 11.6, in cui la sezione di file non rientra nei confini di una pagina: in tal caso verrà 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.



**Figura 11.7:** Schema della mappatura in memoria di file di dimensioni inferiori alla lunghezza richiesta.

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. 11.7.

Non tutti i file possono venire mappati in memoria, dato che, come illustrato in fig. 11.5, 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. 4.2.2). 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, 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. 5.2.4). 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 standard dei file di cap. 6. 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 standard 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 file attraverso l'interfaccia standard, quando lo si è mappato in memoria, è invece possibile usare l'interfaccia standard 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:

```
#include <unistd.h>
#include <sys/mman.h>
int msync(const void *start, size_t length, int flags)
    Sincronizza i contenuti di una sezione di un file mappato in memoria.
```

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

<b>EINVAL</b>	O <code>start</code> non è multiplo di <code>PAGESIZE</code> , o si è specificato un valore non valido per <code>flags</code> .
<b>EFAULT</b>	L'intervallo specificato non ricade in una zona precedentemente mappata.

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 standard troveranno un contenuto del file aggiornato.

Valore	Significato
<code>MS_ASYNC</code>	Richiede la sincronizzazione.
<code>MS_SYNC</code>	Attende che la sincronizzazione si eseguita.
<code>MS_INVALIDATE</code>	Richiede che le altre mappature dello stesso file siano invalidate.

*Tabella 11.4:* Valori dell'argomento `flag` di `msync`.

L'argomento `flag` è specificato come maschera binaria composta da un OR dei valori riportati in tab. 11.4, 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 memoria virtuale, ritornando subito, mentre con il secondo attende che la sincronizzazione sia stata effettivamente eseguita. Il terzo flag fa invalidare le pagine di cui si richiede la sincronizzazione per tutte le mappature dello stesso file, così che esse possano essere immediatamente aggiornate ai 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 <unistd.h>
#include <sys/mman.h>
int munmap(void *start, size_t length)
    Rilascia la mappatura sulla sezione di memoria specificata.
```

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

<b>EINVAL</b>	L'intervallo specificato non ricade in una zona precedentemente mappata.
---------------	--

La funzione cancella la mappatura per l'intervallo specificato attraverso `start` e `length`, ed ogni successivo accesso a tale regione causerà un errore di accesso in memoria. L'argomento `start` deve essere allineato alle dimensioni di una pagina di memoria, e la mappatura di tutte le pagine contenute (anche parzialmente) nell'intervallo indicato, verrà rimossa. Indicare un intervallo che non contiene pagine mappate non è un errore.

Alla conclusione del processo, ogni pagina mappata verrà automaticamente rilasciata, mentre la chiusura del file descriptor usato per effettuare la mappatura in memoria non ha alcun effetto sulla stessa.

## 11.4 Il file locking

In sez. 6.3.1 abbiamo preso in esame le modalità in cui un sistema unix-like gestisce la condivisione dei 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 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 scrittura.

### 11.4.1 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*,<sup>22</sup> 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. 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*:<sup>23</sup> 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.

Richiesta	Stato del file		
	Nessun lock	Read lock	Write lock
Read lock	SI	SI	NO
Write lock	SI	NO	NO

**Tabella 11.5:** Tipologie di file locking.

<sup>22</sup>Stevens in [1] fa riferimento a questo argomento come al *record locking*, dizione utilizzata anche dal manuale delle *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*.

<sup>23</sup>di 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*).

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 standardizzata da POSIX.1 (derivata da System V), che è basata sulla funzione `fcntl`. I *file lock* sono implementati in maniera completamente indipendente nelle due interfacce, 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 lock 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 lock. La situazione delle varie possibilità è riassunta in tab. 11.5, dove si sono riportati, per le varie tipologie di lock presenti su un file, il risultato che si ha in corrispondenza alle due tipologie di *file lock* menzionate, nel successo della richiesta.

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 lock 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).

#### 11.4.2 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 usata per richiedere e rimuovere un *file lock* è `flock`, ed il suo prototipo è:

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Applica o rimuove un *file lock* sul file `fd`.

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

`EWOULDBLOCK` Il file ha già un blocco attivo, e si è specificato `LOCK_NB`.

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 utilizzando le costanti riportate in tab. 11.6.

Valore	Significato
<code>LOCK_SH</code>	Asserisce uno <i>shared lock</i> sul file.
<code>LOCK_EX</code>	Asserisce un <i>exclusive lock</i> sul file.
<code>LOCK_UN</code>	Rilascia il <i>file lock</i> .
<code>LOCK_NB</code>	Impedisce che la funzione si blocchi nella richiesta di un <i>file lock</i> .

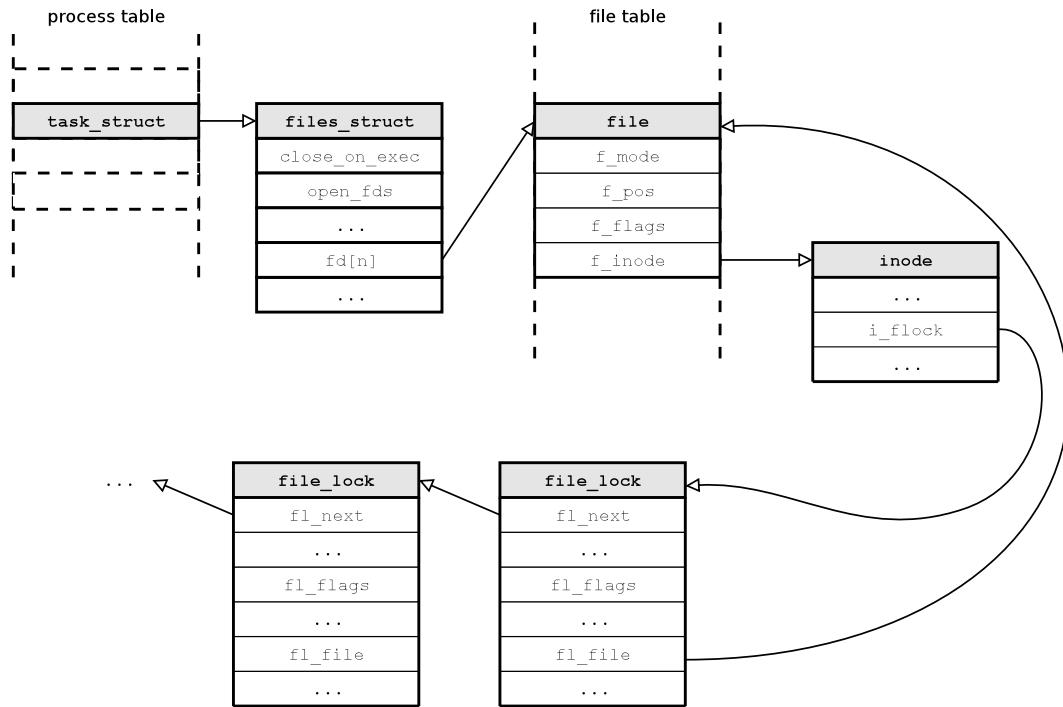
**Tabella 11.6:** Valori dell'argomento `operation` di `flock`.

I primi due valori, `LOCK_SH` e `LOCK_EX` permettono di richiedere un *file lock*, ed ovviamente devono essere usati in maniera alternativa. Se si specifica anche `LOCK_NB` la funzione non si bloccherà qualora il lock non possa essere acquisito, ma ritornerà subito con un errore di `EWOULDBLOCK`. Per rilasciare un lock si dovrà invece usare `LOCK_UN`.

La semantica del file locking di BSD è diversa da quella del file locking POSIX, in particolare per quanto riguarda il comportamento dei lock nei confronti delle due funzioni `dup` e `fork`. Per capire queste differenze occorre descrivere con maggiore dettaglio come viene realizzato il file locking nel kernel in entrambe le interfacce.

In fig. 11.8 si è riportato uno schema essenziale dell'implementazione del file locking in stile BSD in Linux; il punto fondamentale da capire è che un lock, qualunque sia l'interfaccia

che si usa, anche se richiesto attraverso un file descriptor, agisce sempre su un file; perciò le informazioni relative agli eventuali *file lock* sono mantenute a livello di inode,<sup>24</sup> dato che questo è l'unico riferimento in comune che possono avere due processi diversi che aprono lo stesso file.



**Figura 11.8:** Schema dell'architettura del file locking, nel caso particolare del suo utilizzo da parte della 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.<sup>25</sup> Nel caso dei lock 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. 11.8, associando ad ogni nuovo *file lock* un puntatore<sup>26</sup> alla voce nella *file table* da cui si è richiesto il lock, che così ne identifica il titolare.

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 lock. Allora se ricordiamo quanto visto in sez. 6.3.4 e sez. 6.3.1, 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,<sup>27</sup> o se si esegue la rimozione in un processo figlio; inoltre una volta tolto un file lock,

<sup>24</sup>in particolare, come accennato in fig. 11.8, 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 `fs.h` nei sorgenti del kernel). Un bit del campo `fl_flags` specifica se si tratta di un lock in semantica BSD (`FL_FLOCK`) o POSIX (`FL_POSIX`).

<sup>25</sup>cioè una nuova struttura `file_lock`.

<sup>26</sup>il puntatore è mantenuto nel campo `fl_file` di `file_lock`, e viene utilizzato solo per i lock creati con la semantica BSD.

<sup>27</sup>attenzione, 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.

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 su processi diversi.

Infine, per evitare che la terminazione imprevista di un processo lasci attivi dei file lock, quando un file viene chiuso il kernel provveda anche a rimuovere tutti i lock 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 lock rimosso) fintanto 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 lock non viene rilasciato.

Si tenga presente infine che `flock` non è in grado di funzionare per i file mantenuti su NFS, in questo caso, se si ha la necessità di eseguire il *file locking*, occorre usare l'interfaccia basata su `fcntl` che può funzionare anche attraverso NFS, a condizione che sia il client che il server supportino questa funzionalità.

#### 11.4.3 Il file locking POSIX

La seconda interfaccia per l'*advisory locking* disponibile in Linux è quella standardizzata da POSIX, basata sulla funzione `fcntl`. Abbiamo già trattato questa funzione nelle sue molteplici possibilità di utilizzo in sez. 6.3.5. Quando la si impiega per il *file locking* essa viene usata solo secondo il prototipo:

```
#include <fcntl.h>
int fcntl(int fd, int cmd, struct flock *lock)
    Applica o rimuove un file lock sul file fd.
```

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

<code>EACCES</code>	L'operazione è proibita per la presenza di <i>file lock</i> da parte di altri processi.
<code>ENOLCK</code>	Il sistema non ha le risorse per il locking: ci sono troppi segmenti di lock aperti, si è esaurita la tabella dei lock, o il protocollo per il locking remoto è fallito.
<code>EDEADLK</code>	Si è richiesto un lock su una regione bloccata da un altro processo che è a sua volta in attesa dello sblocco di un lock mantenuto dal processo corrente; si avrebbe pertanto un <i>deadlock</i> . Non è garantito che il sistema riconosca sempre questa situazione.
<code>EINTR</code>	La funzione è stata interrotta da un segnale prima di poter acquisire un lock. ed inoltre <code>EBADF</code> , <code>EFAULT</code> .

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 lock preesistenti. Per poter fare tutto questo la funzione utilizza come terzo argomento una apposita struttura `flock` (la cui definizione è riportata in fig. 11.9) nella quale inserire tutti i dati relativi ad un determinato lock. Si tenga presente poi che un 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 11.9:* 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 lock: `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 semanticità dell'omonimo argomento di `lseek`, coi tre possibili valori `SEEK_SET`, `SEEK_CUR` e `SEEK_END`, (si vedano le relative descrizioni in sez. 6.2.3).

Si tenga presente che un 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.

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 file lock.

**Tabella 11.7:** Valori possibili per il campo `l_type` di `flock`.

Il tipo di file lock richiesto viene specificato dal campo `l_type`, esso può assumere i tre valori definiti dalle costanti riportate in tab. 11.7, che permettono di richiedere rispettivamente uno *shared lock*, un *exclusive lock*, e la rimozione di un lock 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 lock.

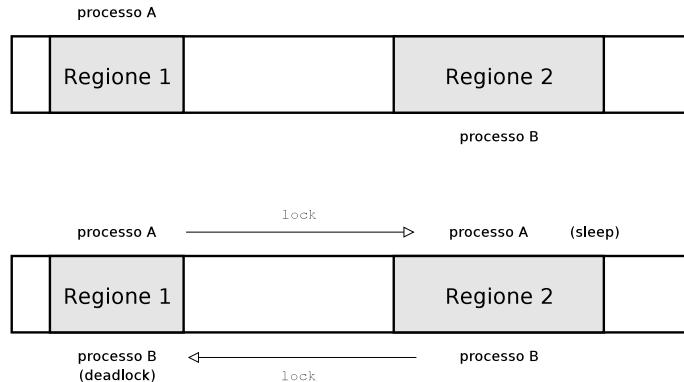
Oltre a quanto richiesto tramite i campi di `flock`, l'operazione effettivamente svolta dalla funzione è stabilita dal valore dall'argomento `cmd` che, come già riportato in sez. 6.3.5, specifica l'azione da compiere; i valori relativi al file locking sono tre:

- `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 lock 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 lock 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 lock, mette il processo in stato di attesa fintanto che il lock 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 lock 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 lock (se è `F_WRLCK`) o di write lock (se è `F_RDLCK`). Si consideri poi che può esserci più di un lock 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 lock fra le due chiamate) per cui si deve sempre verificare il codice di ritorno di `fcntl`<sup>28</sup> quando la si invoca con `F_SETLK`, per controllare che il lock sia stato effettivamente acquisito.



**Figura 11.10:** 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. 11.10, in cui il processo A blocca la regione 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 rilascia 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 lock 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. 11.4.2) esaminiamo più in dettaglio come viene gestito dal kernel. Lo schema delle strutture utilizzate è riportato in fig. 11.11; come si vede esso è molto simile all’analogo di fig. 11.8:<sup>29</sup> il lock 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.

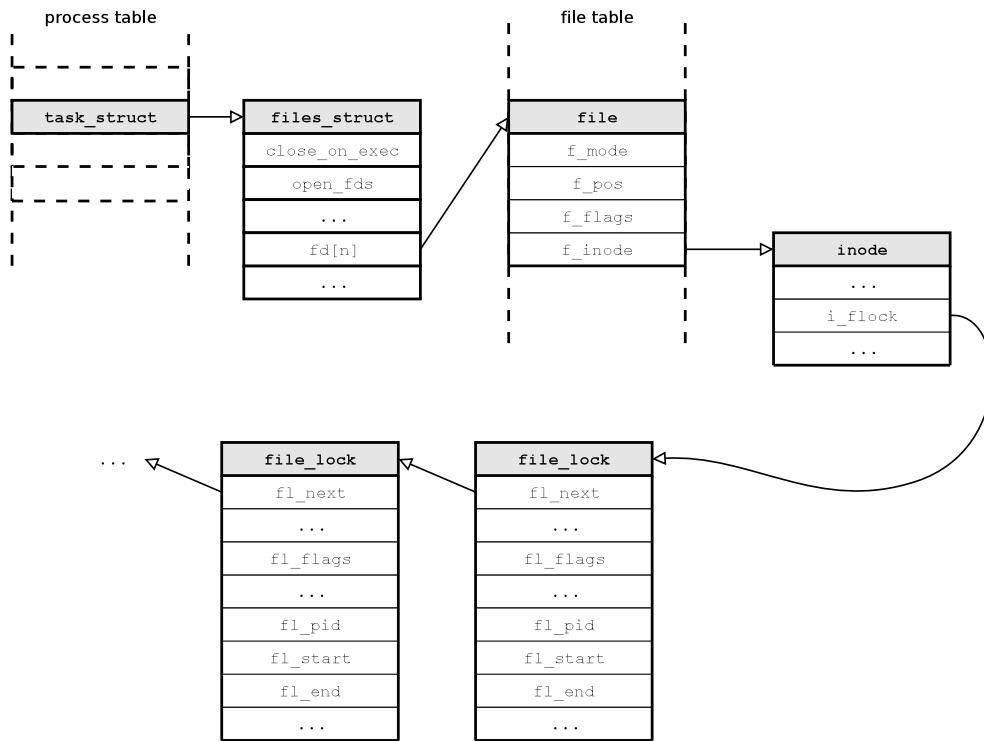
Quando si richiede un lock il kernel effettua una scansione di tutti i lock presenti sul file<sup>30</sup> per verificare se la regione richiesta non si sovrappone ad una già bloccata, in caso affermativo decide in base al tipo di lock, in caso negativo il nuovo lock viene comunque acquisito ed aggiunto alla lista.

Nel caso di rimozione invece questa viene effettuata controllando che il *pid* del processo richiedente corrisponda a quello contenuto nel lock. Questa diversa modalità ha delle conseguenze precise riguardo il comportamento dei lock POSIX. La prima conseguenza è che un 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 processo termina tutti i file lock da esso detenuti vengono immediatamente rilasciati.

<sup>28</sup>controllare 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.

<sup>29</sup>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 `f1_pid`, la sezione di file che viene bloccata grazie ai campi `f1_start` e `f1_end`. La struttura è comunque la stessa, solo che in questo caso nel campo `f1_flags` è impostato il bit `FL_POSIX` ed il campo `f1_file` non viene usato.

<sup>30</sup>scandisce cioè la linked list delle strutture `file_lock`, scartando automaticamente quelle per cui `f1_flags` non è `FL_POSIX`, così che le due interfacce restano ben separate.



**Figura 11.11:** Schema dell’architettura del file locking, nel caso particolare del suo utilizzo secondo l’interfaccia standard POSIX.

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 lock, 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 lock ad esso associati vengono rimossi, nella semantica POSIX basterà chiudere un file descriptor qualunque per cancellare tutti i lock 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 lock 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 lock 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 caso della semantica BSD, essendo i lock relativi a tutto un file e non accumulandosi,<sup>31</sup> la cosa non ha alcun effetto; la funzione ritorna con successo, senza che il kernel debba modificare la lista dei lock. In questo caso invece si possono avere una serie di situazioni diverse: ad esempio è possibile rimuovere con una sola chiamata più lock distinti (indicando in una regione che si sovrapponga completamente a quelle di questi ultimi), o rimuovere solo una parte di un lock preesistente (indicando una regione contenuta in quella di un altro lock), creando un buco, o coprire con un nuovo lock altri lock già ottenuti, e così via, a seconda di come si sovrappongono le regioni richieste e del tipo di operazione richiesta. Il comportamento seguito in questo caso 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 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

<sup>31</sup>questa ultima caratteristica è vera in generale, se cioè si richiede più volte lo stesso file lock, o più lock sulla stessa sezione di file, le richieste non si cumulano e basta una sola richiesta di rilascio per cancellare il lock.

---

```

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_SETLK; /* 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 use 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 11.12:* Sezione principale del codice del programma *Flock.c*.

una sezione di un file usando la semantica POSIX, o un intero file usando la semantica BSD; in fig. 11.12 è riportata il corpo principale del codice del programma, (il testo completo è allegato nella directory dei sorgenti).

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 lock (bloccante o meno), a seconda dell'opzione `-b`.

Il programma inizia col controllare (11-14) che venga passato un parametro (il file da bloccare), che sia stato scelto (15-18) il tipo di lock, 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_SETLKW` 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 lock 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 esegue (41).

In entrambi i casi dopo aver richiesto il lock 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 lock vengono rilasciati.

Con il programma possiamo fare varie verifiche sul funzionamento del file locking; cominciamo con l'eseguire un read lock su un file, ad esempio usando all'interno di un terminale il seguente comando:

```
[piccardi@gont sources]$ ./flock -r Flock.c
Lock acquired
```

il programma segnalera di aver acquisito un lock 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 lock, 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 lock 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 lock 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 -l120 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 -l110 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 -l110 Flock.c
```

il programma cioè si bloccherà nella chiamata a `fcntl`; se a questo punto rilasciamo il precedente lock (terminando il primo comando un C-c sul terminale) potremo verificare che sull'altro terminale il lock viene acquisito, con la comparsa di una nuova riga:

```
[piccardi@gont sources]$ ./flock -w -s0 -l110 Flock.c
Lock acquired
```

Un'altra cosa che si può controllare con il nostro programma è l'interazione fra i due tipi di lock; se ripartiamo dal primo comando con cui si è ottenuto un lock in lettura sull'intero file, possiamo verificare cosa succede quando si cerca di ottenere un lock in scrittura con la semantica BSD:

```
[root@gont sources]# ./flock -f -w Flock.c
Lock acquired
```

che ci mostra come i due tipi di lock 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 lock applicati con l'altra non avrebbero nessun effetto.

#### 11.4.4 La funzione `lockf`

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 (ripresa da System V) che utilizza la funzione `lockf`, il cui prototipo è:

```
#include <sys/file.h>
int lockf(int fd, int cmd, off_t len)
Applica, controlla o rimuove un file lock sul file fd.
```

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

**EWOULDBLOCK** Non è possibile acquisire il lock, e si è selezionato `LOCK_NB`, oppure l'operazione è proibita perché il file è mappato in memoria.

**ENOLCK** Il sistema non ha le risorse per il locking: ci sono troppi segmenti di lock aperti, si è esaurita la tabella dei lock.

ed inoltre **EBADF**, **EINVAL**.

Il comportamento della funzione dipende dal valore dell'argomento `cmd`, che specifica quale azione eseguire; i valori possibili sono riportati in tab. 11.8.

Qualora il lock non possa essere acquisito, a meno di non aver specificato `LOCK_NB`, la funzione si blocca fino alla disponibilità dello stesso. Dato che la funzione è implementata utilizzando

Valore	Significato
LOCK_SH	Richiede uno <i>shared lock</i> . Più processi possono mantenere un lock condiviso sullo stesso file.
LOCK_EX	Richiede un <i>exclusive lock</i> . Un solo processo alla volta può mantenere un lock esclusivo su un file.
LOCK_UN	Sblocca il file.
LOCK_NB	Non blocca la funzione quando il lock non è disponibile, si specifica sempre insieme ad una delle altre operazioni con un OR aritmetico dei valori.

**Tabella 11.8:** Valori possibili per l'argomento cmd di lockf.

`fcntl` la semantica delle operazioni è la stessa di quest'ultima (pertanto la funzione non è affatto equivalente a `flock`).

#### 11.4.5 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`. Se si ricorda quanto esposto in sez. 5.3.2), esso viene di norma utilizzato per cambiare il group-ID 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*.<sup>32</sup>

L'uso del *mandatory locking* presenta vari aspetti delicati, dato che neanche root può passare sopra ad un lock; pertanto un processo che blocchi un file cruciale può renderlo completamente inaccessibile, rendendo completamente inutilizzabile il sistema<sup>33</sup> inoltre con il *mandatory locking* si può bloccare completamente un server NFS richiedendo una lettura su un file su cui è attivo un lock. 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 tab. 8.9, o con l'opzione `mand` per il comando).

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 lock richiesti con l'interfaccia di `flock`, e che la granularità del lock è quella del singolo byte, come per `fcntl`.

La sintassi di acquisizione dei lock 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 lock, a meno di non aver aperto il file con `O_NONBLOCK`, nel qual caso essa ritornerà immediatamente con un errore di `EAGAIN`.

<sup>32</sup>un lettore attento potrebbe ricordare quanto detto in sez. 5.3.6 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*.

<sup>33</sup>il problema si potrebbe risolvere rimuovendo il bit `sgid`, ma non è detto che sia così facile fare questa operazione con un sistema bloccato.

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 lock, 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 lock (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 (che abbiamo trattato in sez. 11.3.2); 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 lock<sup>34</sup> in Linux è stata però fatta la scelta implementativa<sup>35</sup> 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.

---

<sup>34</sup>alcuni 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.

<sup>35</sup>per i dettagli si possono leggere le note relative all'implementazione, mantenute insieme ai sorgenti del kernel nel file `Documentation/mandatory.txt`.

# Capitolo 12

## La comunicazione 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.

Tralasceremo 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*) e CORBA (*Common Object Request Broker Architecture*) che in genere sono implementati con un ulteriore livello sopra i meccanismi elementari.

### 12.1 La comunicazione 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 particolar 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.

#### 12.1.1 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<sup>1</sup> connessi fra di loro in modo che se quanto scrive su di uno si può rileggere dall’altro. Si viene così a costituire un canale di comunicazione tramite i due file descriptor, nella forma di un *tubo* (da cui il nome) attraverso cui fluiscono i dati.

La funzione 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])
```

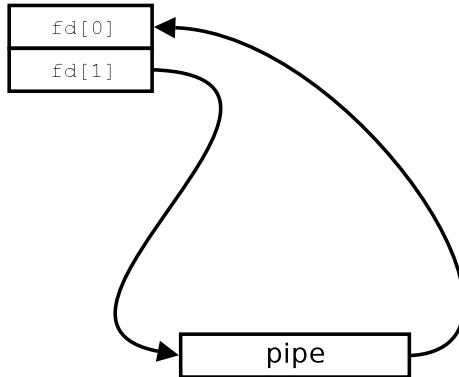
Crea una coppia di file descriptor associati ad una *pipe*.

La funzione restituisce zero in caso di successo e -1 per un errore, nel qual caso **errno** potrà assumere i valori **EMFILE**, **ENFILE** e **EFAULT**.

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,

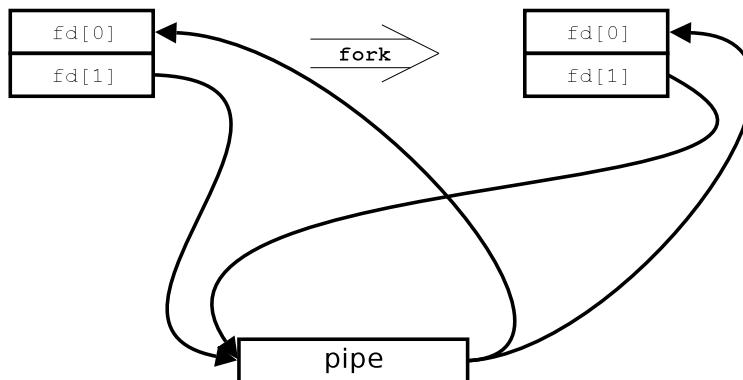
<sup>1</sup>si tenga presente che le *pipe* sono oggetti creati dal kernel e non risiedono su disco.

ma ad un buffer nel kernel, la cui dimensione è specificata dal parametro di sistema PIPE\_BUF, (vedi sez. 8.1.3). Lo schema di funzionamento di una pipe è illustrato in fig. 12.1, in cui sono illustrati i due capi della pipe, associati a ciascun file descriptor, con le frecce che indicano la direzione del flusso dei dati.



**Figura 12.1:** Schema della struttura di una pipe.

Chiaramente creare una pipe all'interno di un singolo processo non serve a niente; se però ricordiamo quanto esposto in sez. 6.3.1 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. 12.2). In questo modo se uno dei processi scrive su un capo della pipe, l'altro può leggere.



**Figura 12.2:** 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<sup>2</sup> limite nell'uso delle pipe. 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

<sup>2</sup>Stevens in [1] riporta come limite anche il fatto che la comunicazione è unidirezionale, ma in realtà questo è un limite facilmente superabile usando una coppia di pipe.

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.

### 12.1.2 Un esempio dell'uso delle pipe

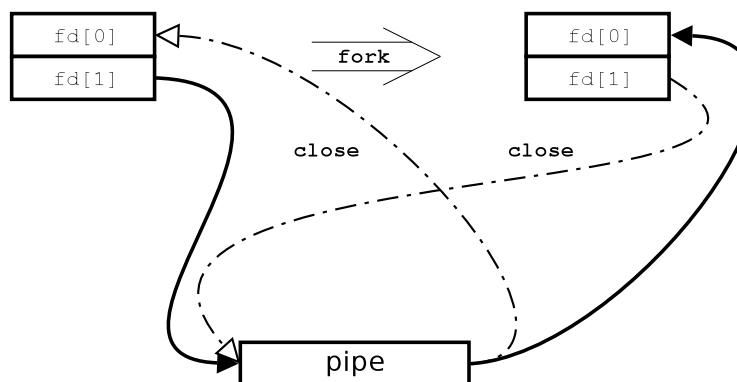
Per capire meglio il funzionamento delle pipe faremo un esempio di quello che è il loro uso più comune, analogo a quello effettuato della shell, e che consiste nell'inviare l'output di un processo (lo standard output) sull'input di un'altro. Realizzeremo il programma di esempio nella forma di un *CGI*<sup>3</sup> per Apache, che genera una immagine JPEG di un codice a barre, specificato come parametro di input.

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?parametro
```

ed il risultato dell'elaborazione deve essere presentato (con una intestazione che ne descrive il mime-type) sullo standard output, in modo che il web-server possa reinviarla al browser che ha effettuato la richiesta, che in questo modo è in grado di visualizzarla opportunamente.

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 del secondo, secondo lo schema mostrato in fig. 12.3, in cui la direzione del flusso dei dati è data dalle frecce continue.



**Figura 12.3:** 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.

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* deve poter gestire più richieste in concorrenza, e si avrebbe una evidente *race condition* in caso di accesso simultaneo a detto

<sup>3</sup>Un *CGI* (*Common Gateway Interface*) è un programma che permette la creazione dinamica di un oggetto da inserire all'interno di una pagina HTML.

file.<sup>4</sup> 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. 6.3.4, in particolare di `dup2`. È attraverso queste funzioni infatti che è possibile dirottare gli stream standard dei processi (che abbiamo visto in sez. 6.1.2 e sez. 7.1.3) sulla pipe. In fig. 12.4 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.<sup>5</sup>

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 redirezione 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 input) e poi scrive (27) la stringa da convertire sul capo in output, 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 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 gli appositi switch 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

---

<sup>4</sup>il problema potrebbe essere superato determinando in anticipo un nome appropriato per il file temporaneo, che verrebbe utilizzato dai vari sotto-processi, e cancellato alla fine della loro esecuzione; ma a questo le cose non sarebbero più tanto semplici.

<sup>5</sup>la funzione `WriteMess` non è riportata in fig. 12.4; 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.

---

```

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 12.4:* Sezione principale del codice del *CGI BarCodePage.c*.

ulteriori dati in ingresso (l'unico modo che un programma ha per sapere che l'input è terminato è rilevare che lo standard input è stato chiuso), e la `wait` non ritornerebbe.

### 12.1.3 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 il programma command, di cui, a seconda di type, restituisce, lo standard input o lo
    standard output nella pipe collegata allo stream restituito come valore di ritorno.
```

La funzione restituisce 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`, ed 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 indicare se la pipe sarà collegata allo standard input o allo standard output del comando invocato.

La funzione restituisce il puntatore allo 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`.

Lo stream restituito da `popen` è identico a tutti gli effetti ai file stream visti in cap. 7, anche se è collegato ad una pipe e non ad un file, e viene sempre aperto in modalità *fully-buffered* (vedi sez. 7.1.4); 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 il file stream, restituito da una precedente popen attendendo la terminazione del
    processo ad essa associato.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore; nel quel caso il valore di `errno` deriva dalle sottostanti chiamate.

che oltre alla chiusura dello stream si incarica anche di attendere (tramite `wait4`) la conclusione del processo creato dalla precedente `popen`.

Per illustrare l'uso di queste due funzioni riprendiamo il problema precedente: il programma mostrato in fig. 12.4 per quanto funzionante, è (volutamente) codificato in maniera piuttosto complessa, inoltre nella pratica sconta un problema di `gs` che non è in grado<sup>6</sup> di riconoscere correttamente l'Encapsulated PostScript, per cui deve essere usato il PostScript e tutte le volte viene generata 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 lo switch `-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 funziona, 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,<sup>7</sup> dal quale poi si

<sup>6</sup>nella versione GNU Ghostscript 6.53 (2002-02-13).

<sup>7</sup>il *Portable PixMap file format* è un formato usato spesso come formato intermedio per effettuare conversioni,

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'output di ciascuno all'input 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 il suo output sullo standard input del successivo, occorrerà usare `popen` aprendo la pipe in scrittura. Il codice del nuovo programma è riportato in fig. 12.5. 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 ciascun processo, per quanto lanciato per primo, si bloccherà in attesa di ricevere sullo standard input il risultato dell'elaborazione del precedente, benchè quest'ultimo venga invocato dopo.

---

```

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\u002dwhite\u002d10",
8         "pnmcrop",
9         "gs\u002dDEVICE=ppmraw\u002d outputFile=-\u002d sNOPAUSE\u002dquiet\u002d-c\u002dshowpage\u002d-c\u002dquit"
10    };
11    char content []="Content-type:\u002dimage/png\u002dn\u002dn";
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 12.5:* Codice completo del *CGI BarCode.c*.

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 redirezione.

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.

---

è infatti molto facile da manipolare, dato che usa caratteri ASCII per memorizzare le immagini, anche se per questo è estremamente inefficiente.

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 redirezione, 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, nell'ordine inverso rispetto a quello in cui le si sono create, tutte le pipe create con `pclose`.

#### 12.1.4 Le *pipe* con nome, o *fifo*

Come accennato in sez. 12.1.1 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 definito dei nuovi oggetti, le *fifo*, che hanno le stesse caratteristiche delle pipe, ma che invece di essere strutture interne del kernel, visibili solo attraverso un file descriptor, sono accessibili attraverso un inode che risiede sul filesystem, 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 apposito buffer nel kernel, senza transitare dal filesystem; l'inode allocato sul filesystem serve infatti solo a fornire un punto di riferimento per i processi, che permetta loro di accedere alla stessa fifo; il comportamento delle funzioni di lettura e scrittura è identico a quello illustrato per le pipe in sez. 12.1.1.

Abbiamo già visto in sez. 5.1.5 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 sarà collegato al capo di uscita della fifo, e dovrà leggere, nel secondo al capo di ingresso, e dovrà scrivere.

Il kernel crea una singola pipe per ciascuna fifo che sia stata aperta, che può 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 di norma la funzione `open` 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,<sup>8</sup> 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 di lettura; è possibile anche usare la fifo all'interno di un solo processo, nel qual caso però occorre stare molto attenti alla possibili situazioni di stallo.<sup>9</sup>

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. 12.1.1).

A parte il caso precedente, che resta probabilmente il più comune, Stevens riporta in [1] 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`).

---

<sup>8</sup>lo standard POSIX lascia indefinito il comportamento in questo caso.

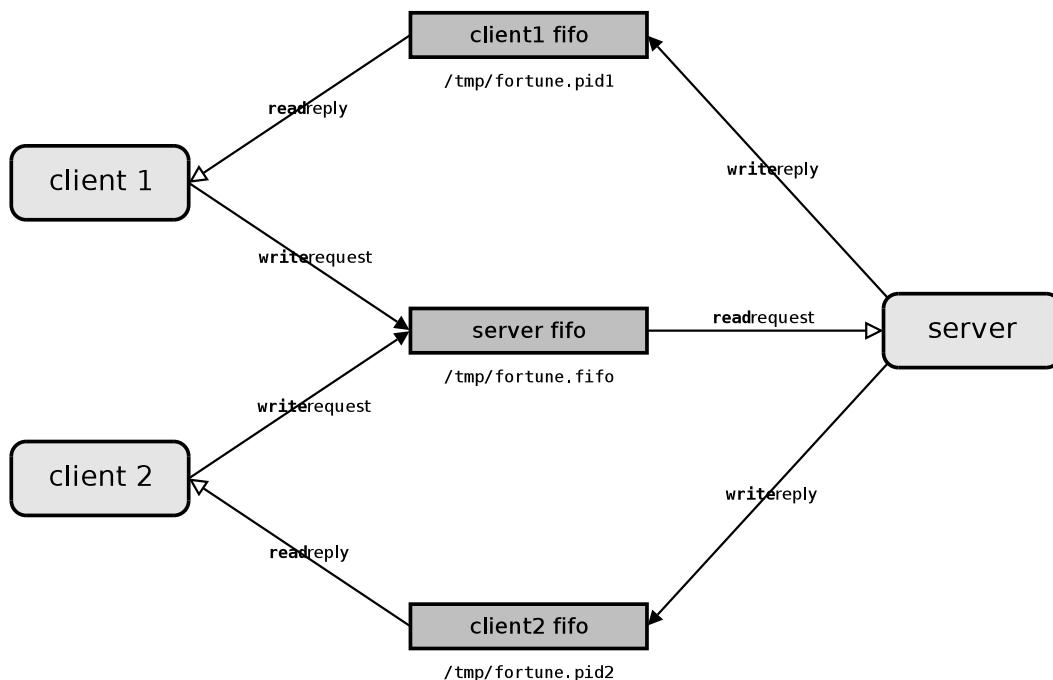
<sup>9</sup>se si cerca di leggere da una fifo che non contiene dati si avrà un deadlock immediato, dato che il processo si blocca e non potrà quindi mai eseguire le funzioni di scrittura.

- Come canale di comunicazione fra client ed 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 i vogliono inviare i dati, questi ultimi saranno stati posti in esecuzione ridirigendo 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 il server ed un numero impreciso 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. 12.6 in cui i client inviano le richieste al server su una fifo nota mentre le risposte vengono reinviata dal server a ciascuno di essi su una fifo temporanea creata per l'occasione.



**Figura 12.6:** 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 importabili da riga di comando. Il corpo principale del server è riportato in fig. 12.7, dove si è tralasciata la parte che tratta la gestione delle opzioni a riga di comando, che effettua il settaggio delle variabili `fortunefilename`, 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`.

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 routine (non riportata) che ne stampa le modalità d'uso. Dopo di che installa (13-15) la funzione

---

```

1 char *fifo_name = "/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(fifo_name, 0622)) { /* create well known fifo if does'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(fifo_name, O_RDONLY);
26    if (fifo_server < 0) {
27        perror("Cannot open read only well known fifo");
28        exit(1);
29    }
30    if (open(fifo_name, 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, fortune[n], strlen(fortune[n])+1); /* write phrase */
49        close(fifo_client); /* close client fifo */
50    }
51 }
52 }
```

---

**Figura 12.7:** Sezione principale del codice del server di *fortunes* basato sulle fifo.

che gestisce i segnali di interruzione (anche questa non è riportata in fig. 12.7) 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 si può chiamare (23) 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 input, ma ritornerà in continuazione, restituendo un end-of-file.<sup>10</sup>

Per questo motivo, dopo aver eseguito l'apertura in lettura (24-28),<sup>11</sup> 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 routine 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 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. 12.8, 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.

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

<sup>10</sup>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.

<sup>11</sup>di solito si effettua l'apertura del capo in lettura di una fifo in modalità non bloccante, per evitare il rischio di uno stall: 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.

---

```

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!=EXIST) {
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 12.8:** Sezione principale del codice del client di *fortunes* basato sulle fifo.

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*, per poter usare quest'ultima occorrerà definire la speciale 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. 10.1.5):

```
[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 inviare un segnale di terminazione con `killall fortuned` 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 inconvenienti<sup>12</sup>; 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.

### 12.1.5 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 l'argomento dei *socket* in cap. 14,<sup>13</sup> nell'ambito dell'interfaccia generale che essi

<sup>12</sup>lo stesso Stevens, che esamina questa architettura in [1], 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).

<sup>13</sup>si tratta comunque di oggetti di comunicazione che, come le pipe, sono utilizzati attraverso dei file descriptor.

forniscono per la programmazione di rete; e vedremo anche (in sez. 14.3.4) come si possono definire dei file speciali (di tipo *socket*, analoghi a quello associati alle fifo) cui si accede però attraverso quella medesima interfaccia; vale però la pena esaminare qui una modalità di uso dei socket locali<sup>14</sup> che li rende sostanzialmente identici ad una pipe bidirezionale.

La funzione **socketpair** infatti consente di creare una coppia di file descriptor connessi fra di 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 restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso **errno** assumerà uno dei valori:

**EAFNOSUPPORT** I socket locali non sono supportati.

**EPROTONOSUPPORT** Il protocollo specificato non è supportato.

**EOPNOTSUPP** Il protocollo specificato non supporta la creazione di coppie di socket.

ed inoltre **EMFILE**, **EFAULT**.

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. I parametri **domain**, **type** e **protocol** derivano dall'interfaccia dei socket (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.

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 linea 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. ??).

## 12.2 Il sistema di comunicazione 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, 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*).

### 12.2.1 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

---

<sup>14</sup>la funzione **socketpair** è stata introdotta in BSD4.4, ma è supportata in genere da qualunque sistema che fornisca l'interfaccia dei socket.

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 che, dato che non c'è, come per i file, un contatore del numero di riferimenti che ne indichi l'essere in uso, essi possono essere cancellati anche se ci sono dei processi che li stanno utilizzando, con tutte le conseguenze (negative) 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. 12.2.3). 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.<sup>15</sup> Oltre la chiave, la struttura, la cui definizione è riportata in fig. 12.9, mantiene varie proprietà ed informazioni associate all'oggetto.

---

```
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 12.9:* La struttura `ipc_perm`, come definita in `sys/ipc.h`.

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 *parenti* 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 parametro attraverso una `exec`.

Però quando i processi non sono *parenti* (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 l'identificatore su un file noto, ma questo ovviamente comporta lo svantaggio di doverselo 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

<sup>15</sup>in 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*.

apposita, `ftok`, che permette di ottenere una chiave specificando il nome di un file ed un numero di versione; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id)
    Restituisce una chiave per identificare un oggetto del SysV IPC.
```

La funzione restituisce la chiave in caso di successo e -1 altrimenti, nel qual caso `errno` sarà uno dei possibili codici di errore di `stat`.

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.<sup>16</sup>

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 ne è stata effettuata una revisione completa nello standard POSIX.1b, che tratteremo in sez. 12.4.

### 12.2.2 Il controllo di accesso

Oltre alle chiavi, abbiamo visto 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. 5.3.1).

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. 5.4<sup>17</sup> e come per i file definiscono gli accessi per il proprietario, il suo gruppo e tutti gli altri.

<sup>16</sup>nelle libc4 e libc5, come avviene in SunOS, l'argomento `proj_id` è dichiarato tipo `char`, le glibc usano il prototipo specificato da XPG4, ma vengono lo stesso utilizzati gli 8 bit meno significativi.

<sup>17</sup>se però si vogliono usare le costanti simboliche ivi definite occorrerà includere il file `sys/stat.h`, alcuni sistemi definiscono le costanti `MSG_R` (0400) e `MSG_W` (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 settati rispettivamente al valore dell'user-ID e del group-ID 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` specifici 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 l'accesso è sempre consentito.
- se l'user-ID 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 appropriato<sup>18</sup> l'accesso è consentito.
- se il group-ID 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.

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. 5.3.7) non ha alcun significato.

### 12.2.3 Gli identificatori ed il loro utilizzo

L'unico campo di `ipc_perm` del quale non abbiamo ancora parlato è `seq`, che in fig. 12.9 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 porta gli identificatori ad assumere tutti i valori possibili, rendendo molto più lungo il periodo in cui un identificatore può venire riutilizzato.

---

<sup>18</sup>per appropriato si intende che è settato il permesso di scrittura per le operazioni di scrittura e quello di lettura per le operazioni di lettura.

Il sistema dispone sempre di un numero fisso di oggetti di IPC,<sup>19</sup> e per ciascuno di essi 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 l'identificatore al numero di oggetti presenti viene sommato il valore di `seq` moltiplicato per il numero massimo di oggetti di quel tipo,<sup>20</sup> si evita così il riutilizzo degli stessi numeri, e si fa sì che l'identificatore assuma tutti i valori possibili.

---

```

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 12.10:** Sezione principale del programma di test per l'assegnazione degli identificatori degli oggetti di IPC `IPCTestId.c`.

In fig. 12.10 è riportato il codice di un semplice programma di test che si limita a creare un oggetto (specificato a riga di comando), stamparne il numero di identificatore e cancellarlo per un numero specificato di volte. Al solito non si è riportato il codice della gestione delle opzioni a riga di comando, che permette di specificare quante volte effettuare il ciclo `n`, e su quale tipo di oggetto eseguirlo.

La figura non riporta il codice di selezione delle opzioni, che permette di inizializzare i valori

---

<sup>19</sup>fino al kernel 2.2.x questi valori, definiti dalle costanti `MSGMNI`, `SEMMLNI` e `SHMMMLNI`, potevano essere cambiati (come tutti gli altri limiti relativi al *SysV IPC*) solo con una ricompilazione del kernel, andando a modificarne la definizione nei relativi header file. A partire dal kernel 2.4.x è possibile cambiare questi valori a sistema attivo scrivendo sui file `shmmmlni`, `msgmnlni` e `sem` di `/proc/sys/kernel` o con l'uso di `sysctl`.

<sup>20</sup>questo vale fino ai kernel della serie 2.2.x, dalla serie 2.4.x viene usato lo stesso fattore per tutti gli oggetti, esso è dato dalla costante `IPCMMLNI`, definita in `include/linux/ipc.h`, che indica il limite massimo per il numero di tutti oggetti di IPC, ed il cui valore è 32768.

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 un ciclo di 5 volte. Se si lancia il comando si otterrà 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 abbiamo un kernel della serie 2.4.x nel quale non avevamo ancora usato nessuna coda di messaggi. Se ripetiamo il comando otterremo ancora:

```
[piccardi@gont sources]$ ./ipctestid
Identifier Value 163840
Identifier Value 196608
Identifier Value 229376
Identifier Value 262144
Identifier Value 294912
```

che ci mostra come il valore di `seq` sia in effetti una quantità mantenuta staticamente all'interno del sistema.

#### 12.2.4 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, anche se la loro struttura è diversa, ed il loro scopo principale è appunto quello di permettere a processi diversi di scambiarsi dei dati.

La funzione che permette di richiedere al sistema l'identificatore di una coda di messaggi esistente (o di crearne una se questa non esiste) è `msgget`; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int flag)
    Restituisce l'identificatore di una coda di messaggi.
```

La funzione restituisce l'identificatore (un intero positivo) o -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EACCES</code>	Il processo chiamante non ha i privilegi per accedere alla coda richiesta.
<code>EEXIST</code>	Si è richiesta la creazione di una coda che già esiste, ma erano specificati sia <code>IPC_CREAT</code> che <code>IPC_EXCL</code> .
<code>EIDRM</code>	La coda richiesta è marcata per essere cancellata.
<code>ENOENT</code>	Si è cercato di ottenere l'identificatore di una coda di messaggi specificando una chiave che non esiste e <code>IPC_CREAT</code> non era specificato.
<code>ENOSPC</code>	Si è cercato di creare una coda di messaggi quando è stato superato il limite massimo di code ( <code>MSGMNI</code> ).

ed inoltre `ENOMEM`.

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, il processo (ed i suoi eventuali figli) potranno farvi riferimento solo attraverso l'identificatore.

Se invece si specifica un valore diverso da `IPC_PRIVATE`<sup>21</sup> 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.

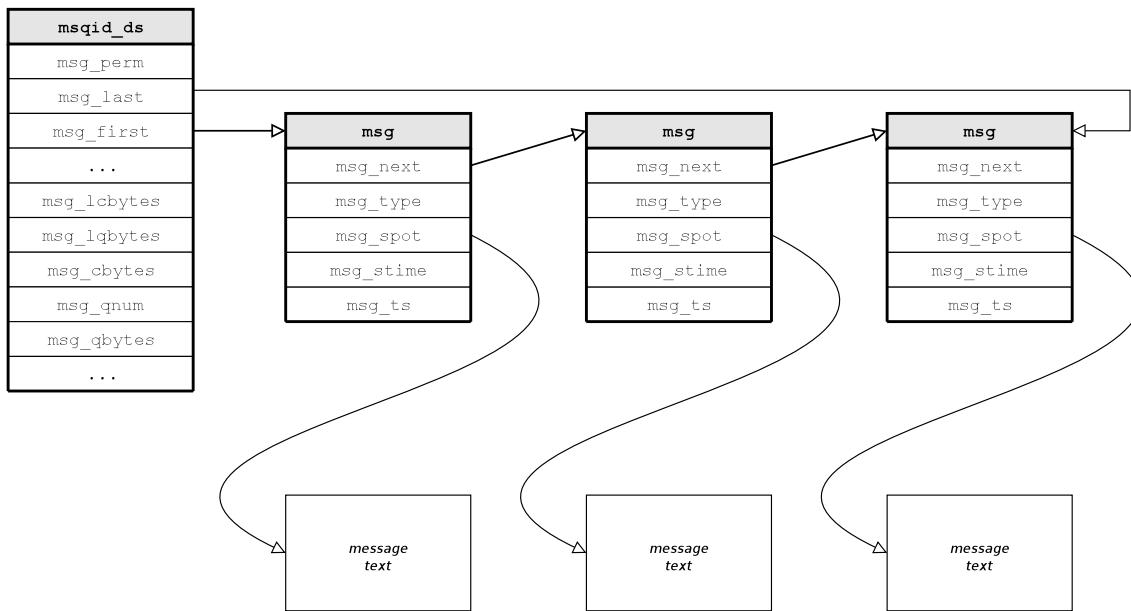
Se invece si vuole creare una nuova coda di messaggi `flag` non può essere nullo e deve essere fornito come maschera binaria, impostando il bit corrispondente al valore `IPC_CREAT`. In questo caso i nove bit meno significativi di `flag` saranno usati come permessi per il nuovo oggetto, secondo quanto illustrato in sez. 12.2.2. 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 una maniera che garantisca l'accesso esclusivo ad una coda di messaggi. Usare `IPC_PRIVATE` o `constIPC_CREAT` e `IPC_EXCL` per `flag` comporta solo la creazione di una nuova coda.

Costante	Valore	File in proc	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>MSGMNB</code>	16384	<code>msgmnb</code>	Dimensione massima del contenuto di una coda.

**Tabella 12.1:** Valori delle costanti associate ai limiti delle code di messaggi.

Le code di messaggi sono caratterizzate da tre limiti fondamentali, definiti negli header e corrispondenti alle prime tre costanti riportate in tab. 12.1, come accennato però in Linux è possibile modificare questi limiti attraverso l'uso di `sysctl` o scrivendo nei file `msgmax`, `msgmnb` e `msgmni` di `/proc/sys/kernel/`.



**Figura 12.11:** Schema della struttura di una coda messaggi.

Una coda di messaggi è costituita da una *linked list*,<sup>22</sup> i nuovi messaggi vengono inseriti in

<sup>21</sup>in Linux questo significa un valore diverso da zero.

<sup>22</sup>una *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

coda alla lista e vengono letti dalla cima, in fig. 12.11 si è riportato lo schema con cui queste strutture vengono mantenute dal kernel.<sup>23</sup>

---

```

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 */
        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() */
        struct msg *msg_first;              /* first message on queue, unused */
        struct msg *msg_last;                /* last message in queue, unused */
        unsigned long int msg_cbytes;        /* current number of bytes on queue */
};


```

---

**Figura 12.12:** La struttura `msqid_ds`, associata a ciascuna coda di messaggi.

A ciascuna coda è associata una struttura `msqid_ds`, la cui definizione, è riportata in fig. 12.12. In questa struttura il kernel mantiene le principali informazioni riguardo lo stato corrente della coda.<sup>24</sup> In fig. 12.12 sono elencati i campi significativi definiti in `sys/msg.h`, a cui si sono aggiunti gli ultimi tre campi che sono previsti dalla implementazione originale di System V, ma non dallo standard Unix98.

Quando si crea una nuova coda con `msgget` questa struttura viene inizializzata, in particolare il campo `msg_perm` viene inizializzato come illustrato in sez. 12.2.2, 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.
- il campo `msg_ctime`, che esprime il tempo di creazione 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`).
- i campi `msg_first` e `msg_last` che esprimono l'indirizzo del primo e ultimo messaggio sono inizializzati a `NULL` e `msg_cbytes`, che esprime la dimensione in byte dei messaggi presenti è inizializzato a zero. Questi campi sono ad uso interno dell'implementazione e non devono essere utilizzati da programmi in user space).

---

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.

<sup>23</sup>lo schema illustrato in fig. 12.11 è in realtà una semplificazione di quello usato effettivamente fino ai kernel della serie 2.2.x, nei kernel della serie 2.4.x la gestione delle code di messaggi è stata modificata ed è effettuata in maniera diversa; abbiamo mantenuto lo schema precedente in quanto illustra comunque in maniera più che adeguata i principi di funzionamento delle code di messaggi.

<sup>24</sup>come accennato questo vale fino ai kernel della serie 2.2.x, essa viene usata nei kernel della serie 2.4.x solo per compatibilità in quanto è quella restituita dalle funzioni dell'interfaccia. Si noti come ci sia una differenza con i campi mostrati nello schema di fig. 12.11 che sono presi dalla definizione di `linux/msg.h`, e fanno riferimento alla definizione della omonima struttura usata nel kernel.

Una volta creata una coda di messaggi le operazioni di controllo vengono effettuate con la funzione `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 l'operazione specificata da cmd sulla coda msqid.
```

La funzione restituisce 0 in caso di successo o -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

- |               |  |
|---------------|--|
| <b>EACCES</b> | Si è richiesto <code>IPC_STAT</code> ma processo chiamante non ha i privilegi di lettura sulla coda.   |
| <b>EIDRM</b>  | La coda richiesta è stata cancellata.  |
| <b>EPERM</b>  | Si è richiesto <code>IPC_SET</code> o <code>IPC_RMID</code> 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 <code>EFAULT</code> ed <code>EINVAL</code> . |

La funzione permette di accedere ai valori della struttura `msqid_ds`, mantenuta all'indirizzo `buf`, per la coda specificata dall'identificatore `msqid`. Il comportamento della funzione dipende dal valore dell'argomento `cmd`, che specifica il tipo di azione da eseguire; i valori possibili sono:

- |                 |  |
|-----------------|--|
| <b>IPC_STAT</b> | Legge le informazioni riguardo la coda nella struttura indicata da <code>buf</code> . Occorre avere il permesso di lettura sulla coda.   |
| <b>IPC_RMID</b> | Rimuove la coda, cancellando tutti i dati, con effetto immediato. Tutti i processi che cercheranno di accedere alla coda riceveranno un errore di <code>EIDRM</code> , 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 user-ID effettivo corrispondente al creatore o al proprietario della coda, o all'amministratore.   |
| <b>IPC_SET</b>  | Permette di modificare i permessi ed il proprietario della coda, ed il limite massimo sulle dimensioni del totale dei messaggi in essa contenuti ( <code>msg_qbytes</code> ). I valori devono essere passati in una struttura <code>msqid_ds</code> puntata da <code>buf</code> . Per modificare i valori di <code>msg_perm.mode</code> , <code>msg_perm.uid</code> e <code>msg_perm.gid</code> occorre essere il proprietario o il creatore della coda, oppure l'amministratore; lo stesso vale per <code>msg_qbytes</code> , ma l'amministratore ha la facoltà di incrementarne il valore a limiti superiori a <code>MSGMNB</code> . |

Una volta che si abbia a disposizione l'identificatore, per inviare un messaggio su una coda si utilizza la funzione `msgsnd`; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsiz, int msgflg)
    Invia un messaggio sulla coda msqid.
```

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

- |               |  |
|---------------|--|
| <b>EACCES</b> | Non si hanno i privilegi di accesso sulla coda.  |
| <b>EIDRM</b>  | La coda è stata cancellata.  |
| <b>EAGAIN</b> | Il messaggio non può essere inviato perché si è superato il limite <code>msg_qbytes</code> sul numero massimo di byte presenti sulla coda, e si è richiesto <code>IPC_NOWAIT</code> in <code>flag</code> . |
| <b>EINTR</b>  | La funzione è stata interrotta da un segnale.  |
| <b>EINVAL</b> | Si è specificato un <code>msgid</code> invalido, o un valore non positivo per <code>mtype</code> , o un valore di <code>msgsz</code> maggiore di <code>MSGMAX</code> .                                     |

ed inoltre `EFAULT` ed `ENOMEM`.

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. 12.13 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`.

La struttura di fig. 12.13 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. 12.13, 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. 12.13, `msgsz` dovrà essere pari a `LENGTH`).

---

```
struct msgbuf {
    long mtype;           /* message type, must be > 0 */
    char mtext[LENGTH];   /* message data */
};
```

---

**Figura 12.13:** Schema della struttura `msgbuf`, da utilizzare come argomento per inviare/ricevere messaggi.

Per capire meglio il funzionamento della funzione riprendiamo in considerazione la struttura della coda illustrata in fig. 12.11. 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 mandando il processo in stato di *sleep*. 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 che viene utilizzata per estrarre un messaggio da una coda è `msgrecv`; il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t msgrecv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int
msgflg)
Legge un messaggio dalla coda msqid.
```

La funzione restituisce il numero di byte letti in caso di successo, e -1 in caso di errore, nel qual caso `errno` assumerà uno dei valori:

<code>EACCES</code>	Non si hanno i privilegi di accesso sulla coda.
<code>EIDRM</code>	La coda è stata cancellata.
<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>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 <code>EFAULT</code> .	

La funzione legge un messaggio dalla coda specificata, scrivendolo sulla struttura puntata da `msgp`, che dovrà avere un formato analogo a quello di fig. 12.13. 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. 12.13).

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. 12.11, 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 in stato di *sleep*. 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_lrpid`, 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; 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. 12.14).

L'altro problema è non facendo uso di file descriptor le tecniche di *I/O multiplexing* descritte in sez. 11.1 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.

In fig. 12.14 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` (8-11) vengono passate le richieste mentre con `msgbuf_write` (12-15) vengono restituite le frasi.

La gestione delle opzioni si è al solito omessa, essa si curerà di impostare in `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 (23) vengono lette 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.

---

```

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 12.14:** Sezione principale del codice del server di *fortunes* basato sulle *message queue*.

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).

---

```

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 12.15:** Sezione principale del codice del client di *fortunes* basato sulle *message queue*.

In fig. 12.15 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 relativa alle strutture usate per la comunicazione tramite le code, sono le stesse viste in fig. 12.14.

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 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 il messaggio di richiesta (12-13) 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 (16) rileggere dalla coda la risposta 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
-- Linuxkongrefß '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 viene 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 problemi sono maggiori, 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.

### 12.2.5 Semafori

I semafori non sono meccanismi di intercomunicazione diretta come quelli (pipe, fifo e code di messaggi) visti finora, e 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. 3.5.2).

Un semaforo è uno speciale contatore, mantenuto nel kernel, che permette, a seconda del suo valore, di consentire o meno la prosecuzione dell'esecuzione di un programma. In questo modo l'accesso ad una risorsa condivisa da più processi può essere controllato, associando ad essa un semaforo che consente di assicurare che non più di un processo alla volta possa usarla.

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 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, siamo invece in una situazione in cui la risorsa non è disponibile, ed il processo si bloccherà in stato di *sleep* fin quando chi la sta utilizzando non la rilascerà, incrementando il valore del semaforo. Non appena il semaforo torna positivo, indicando che la risorsa è disponibile, il processo sarà svegliato, e si 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 comunicazione inter-processo di *SysV IPC* prevede anche i semafori, ma gli oggetti utilizzati non sono semafori singoli, ma gruppi di semafori detti *insiemi* (o *semaphore set*); la funzione che permette di creare o ottenere l’identificatore di un insieme di semafori è `semget`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag)
```

Restituisce l’identificatore di un insieme di semafori.

La funzione restituisce l’identificatore (un intero positivo) o -1 in caso di errore, nel qual caso `errno` assumerà i valori:

<b>ENOSPC</b>	Si è cercato di creare una insieme di semafori quando è stato superato o il limite per il numero totale di semafori ( <b>SEMMNS</b> ) o quello per il numero totale degli insiemi ( <b>SEMMNI</b> ) nel sistema.
<b>EINVAL</b>	L’argomento <b>nsems</b> è minore di zero o maggiore del limite sul numero di semafori per ciascun insieme ( <b>SEMMSL</b> ), o se l’insieme già esiste, maggiore del numero di semafori che contiene.
<b>ENOMEM</b>	Il sistema non ha abbastanza memoria per poter contenere le strutture per un nuovo insieme di semafori.

ed inoltre **EACCES**, **ENOENT**, **EEXIST**, **EIDRM**, con lo stesso significato che hanno per `msgget`.

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. 12.2.4. 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, il problema è che i semafori del *SysV IPC* soffrono di altri due, ben più gravi, difetti.

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.

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; ci si così a trova 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.

---

```
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 12.16:** La struttura `semid_ds`, associata a ciascun insieme di semafori.

A ciascun insieme di semafori è associata una struttura `semid_ds`, riportata in fig. 12.16.<sup>25</sup> 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` viene inizializzato come illustrato in sez. 12.2.2 (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 creazione dell'insieme, viene inizializzato al tempo corrente.
- il campo `sem_otime`, che esprime il tempo dell'ultima operazione effettuata, viene inizializzato a zero.

Ciascun semaforo dell'insieme è realizzato come una struttura di tipo `sem` che ne contiene i dati essenziali, la sua definizione<sup>26</sup> è riportata in fig. 12.17. Questa struttura, non è accessibile in user space, ma i valori in essa specificati possono essere letti in maniera indiretta, attraverso l'uso delle funzioni di controllo.

---

```
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 12.17:** La struttura `sem`, che contiene i dati di un singolo semaforo.

I dati mantenuti nella struttura, ed elencati in fig. 12.17, indicano rispettivamente:

`semval` il valore numerico del semaforo.

`sempid` il *pid* dell'ultimo processo che ha eseguito una operazione sul semaforo.

`semncnt` il numero di processi in attesa che esso venga incrementato.

`semzcnt` il numero di processi in attesa che esso si annulli.

Costante	Valore	Significato
SEMNNI	128	Numero massimo di insiemi di semafori.
SEMMNL	250	Numero massimo di semafori per insieme.
SEMMNS	SEMMNI*SEMMNL	Numero massimo di semafori nel sistema .
SEMVNM	32767	Massimo valore per un semaforo.
SEMOPM	32	Massimo numero di operazioni per chiamata a <code>semop</code> .
SEMMNU	SEMMNS	Massimo numero di strutture di ripristino.
SEMUME	SEMOPM	Massimo numero di voci di ripristino.
SEMAEM	SEMVNM	valore massimo per l'aggiustamento all'uscita.

**Tabella 12.2:** Valori delle costanti associate ai limiti degli insiemi di semafori, definite in `linux/sem.h`.

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. 12.2. Alcuni di

<sup>25</sup>non si sono riportati i campi ad uso interno del kernel, che vedremo in fig. 12.20, che dipendono dall'implementazione.

<sup>26</sup>si è riportata la definizione originaria del kernel 1.0, che contiene la prima realizzazione del *SysV IPC* in Linux. In realtà questa struttura ormai è ridotta ai soli due primi membri, e gli altri vengono calcolati dinamicamente. La si è utilizzata a scopo di esempio, perché indica tutti i valori associati ad un semaforo, restituiti dalle funzioni di controllo, e citati dalle pagine di manuale.

questi limiti sono al solito accessibili e modificabili attraverso `sysctl` o scrivendo direttamente nel file `/proc/sys/kernel/sem`.

La funzione 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:

```
#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 restituisce in caso di successo un valore positivo quanto usata con tre argomenti ed un valore nullo quando usata con quattro. In caso di errore restituisce -1, ed `errno` assumerà uno dei valori:

<code>EACCES</code>	Il processo non ha i privilegi per eseguire l'operazione richiesta.
<code>EIDRM</code>	L'insieme di semafori è stato cancellato.
<code>EPERM</code>	Si è richiesto <code>IPC_SET</code> o <code>IPC_RMID</code> ma il processo non ha privilegi sufficienti ad eseguire l'operazione.
<code>ERANGE</code>	Si è richiesto <code>SETALL</code> <code>SETVAL</code> ma il valore a cui si vuole impostare il semaforo è minore di zero o maggiore di <code>SEMVMX</code> .

ed inoltre `EFAULT` ed `EINVAL`.

La funzione può avere tre o quattro parametri, 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 */
    /* Linux specific part: */
    struct seminfo *__buf;                 /* buffer for IPC_INFO */
};
```

---

**Figura 12.18:** 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 l'argomento esso deve essere passato come una `semun`, la cui definizione, con i possibili valori che può assumere, è riportata in fig. 12.18.

Come già accennato sia il comportamento della funzione che il numero di parametri con cui deve essere invocata, dipendono dal valore dell'argomento `cmd`, che specifica l'azione da intraprendere; i valori validi (che cioè non causano un errore di `EINVAL`) per questo argomento sono i seguenti:

- |                       |   |
|-----------------------|---|
| <code>IPC_STAT</code> | Legge i dati dell'insieme di semafori, copiando il contenuto della relativa struttura <code>semid_ds</code> all'indirizzo specificato con <code>arg.buf</code> . Occorre avere il permesso di lettura. L'argomento <code>semnum</code> viene ignorato.  |
| <code>IPC_RMID</code> | Rimuove l'insieme di semafori e le relative strutture dati, con effetto immediato. Tutti i processi che erano stato di <i>sleep</i> vengono svegliati, ritornando con un errore di <code>EIDRM</code> . L'user-ID 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> . L'user-ID 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>semzcnt</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 sul semaforo. 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 sul semaforo.

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 12.3:* 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. 12.3 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 `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 le operazioni ordinarie su un semaforo o un insieme di semafori.
```

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

<b>EACCES</b>	Il processo non ha i privilegi per eseguire l'operazione richiesta.
<b>EIDRM</b>	L'insieme di semafori è stato cancellato.
<b>ENOMEM</b>	Si è richiesto un <code>SEM_UNDO</code> ma il sistema non ha le risorse per allocare la struttura di ripristino.
<b>EAGAIN</b>	Un'operazione comporterebbe il blocco del processo, ma si è specificato <code>IPC_NOWAIT</code> in <code>sem_flg</code> .
<b>EINTR</b>	La funzione, bloccata in attesa dell'esecuzione dell'operazione, viene interrotta da un segnale.
<b>E2BIG</b>	L'argomento <code>nsops</code> è maggiore del numero massimo di operazioni <code>SEMOPM</code> .
<b>ERANGE</b>	Per alcune operazioni il valore risultante del semaforo viene a superare il limite massimo <code>SEMVMX</code> .
ed inoltre <b>EFAULT</b> ed <b>EINVAL</b> .	

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.

---

```
struct sembuf
{
    unsigned short int sem_num; /* semaphore number */
    short int sem_op;          /* semaphore operation */
    short int sem_flg;         /* operation flag */
};
```

---

**Figura 12.19:** La struttura `sembuf`, usata per le operazioni sui semafori.

Il contenuto di ciascuna operazione deve essere specificato attraverso una opportuna struttura `sembuf` (la cui definizione è riportata in fig. 12.19) 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 in un vettore, per cui il primo semaforo 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, invece di bloccarsi (in tutti quei casi in cui l'esecuzione di una operazione richiede che il processo vada in stato di *sleep*), `semop` ritorni immediatamente con un errore di `EAGAIN`. Impostando `SEM_UNDO` si

richiede invece che l'operazione venga registrata in modo che il valore del semaforo possa essere ripristinato all'uscita del processo.

Infine **sem\_op** è il campo che controlla l'operazione che viene eseguita e determina il comportamento della chiamata a **semop**; tre sono i casi possibili:

**sem\_op > 0** In questo caso il valore di **sem\_op** viene aggiunto al valore corrente di **semval**. La funzione ritorna immediatamente (con un errore di **ERANGE** qualora si sia superato il limite **SEMVMX**) ed il processo non viene bloccato in nessun caso. 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'esecuzione procede immediatamente. Se **semval** è diverso da zero il comportamento è controllato da **sem\_flg**, se è stato impostato **IPC\_NOWAIT** la funzione ritorna con un errore di **EAGAIN**, altrimenti viene incrementato **semzcnt** di uno ed il processo resta in stato di *sleep* fintanto che non si ha una delle condizioni seguenti:

- **semval** diventa zero, nel qual caso **semzcnt** viene decrementato di uno.
- l'insieme di semafori viene rimosso, nel qual caso **semop** ritorna un errore di **EIDRM**.
- il processo chiamante riceve un segnale, nel qual caso **semzcnt** viene decrementato di uno e **semop** ritorna un errore di **EINTR**.

Al processo chiamante è richiesto 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 la funzione ritorna immediatamente; 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** la funzione ritorna 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 un errore di **EIDRM**.
- il processo chiamante riceve un segnale, nel qual caso **semncnt** viene decrementato di uno e **semop** ritorna un errore di **EINTR**.

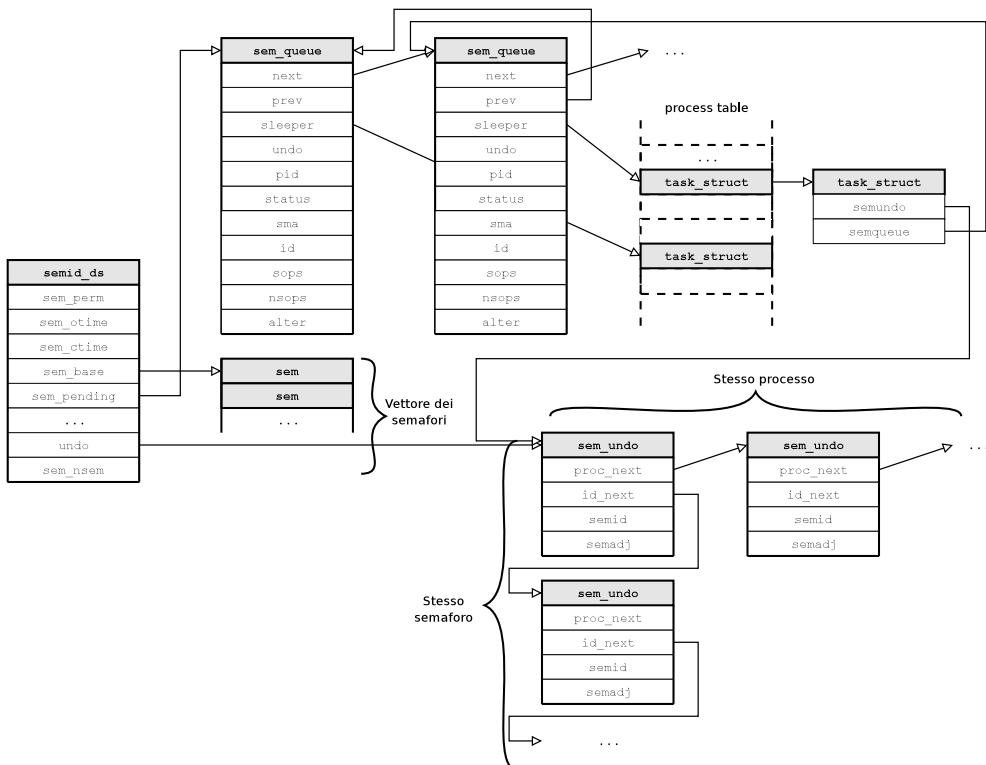
Al processo chiamante è richiesto il privilegio di alterazione (scrittura) sull'insieme di semafori.

In caso di successo della funzione viene aggiornato il campo **sempid** per ogni semaforo modificato al valore del *pid* del processo chiamante; inoltre vengono 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** 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. 12.20. Si è presa come riferimento l'architettura usata fino al kernel 2.2.x che è più semplice (ed illustrata in dettaglio in [10]); nel kernel 2.4.x la struttura del *SysV IPC* è stata modificata, ma le definizioni relative a queste strutture restano per compatibilità.<sup>27</sup>



**Figura 12.20:** Schema della struttura di un insieme di semafori.

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<sup>28</sup>.

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 queste 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`)

<sup>27</sup>in particolare con le vecchie versioni delle librerie del C, come le libc5.

<sup>28</sup>che viene referenziata tramite i campi `sem_pending` e `sem_pending_last` di `semid_ds`.

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 **semadj**) 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,<sup>29</sup> 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.<sup>30</sup> 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 chiamate 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. 12.21. 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*<sup>31</sup> 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

<sup>29</sup>rispettivamente attraverso i due campi **id\_next** e **proc\_next**.

<sup>30</sup>attraverso il campo **semundo** di **task\_struct**, come mostrato in 12.20.

<sup>31</sup>si 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 (library, 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 12.21:** Il codice delle funzioni che permettono di creare o recuperare l'identificatore di un semaforo da utilizzare come *mutex*.

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.

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. 12.3.3 come sia possibile ottenere un'interfaccia analoga a quella appena illustrata, senza incorrere in questi problemi, usando il file locking.

### 12.2.6 Memoria condivisa

Il terzo oggetto introdotto dal *SysV IPC* è quello dei segmenti di memoria condivisa. La funzione che permette di ottenerne uno è `shmget`, ed il suo prototipo è:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int flag)
    Restituisce l'identificatore di una memoria condivisa.
```

La funzione restituisce l'identificatore (un intero positivo) o -1 in caso di errore, nel qual caso `errno` assumerà i valori:

<b>ENOSPC</b>	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.
<b>EINVAL</b>	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.
<b>ENOMEM</b>	Il sistema non ha abbastanza memoria per poter contenere le strutture per un nuovo segmento di memoria condivisa.

ed inoltre `EACCES`, `ENOENT`, `EEXIST`, `EIDRM`, con lo stesso significato che hanno per `msgget`.

La funzione, come `semget`, è del tutto analoga a `msgget`, ed identico è l'uso degli argomenti `key` e `flag` per cui non ripeteremo quanto detto al proposito in sez. 12.2.4. L'argomento `size` specifica invece la dimensione, in byte, del segmento, che viene comunque arrotondata al multiplo superiore di `PAGE_SIZE`.

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.

A ciascun segmento di memoria condivisa è associata una struttura `shmid_ds`, riportata in fig. 12.22. 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. 12.2.2, e valgono le considerazioni ivi fatte relativamente ai permessi di accesso; per quanto riguarda gli altri campi invece:

---

```

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 12.22:** La struttura `shmid_ds`, associata a ciascun segmento di memoria condivisa.

- 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. 12.4 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
<code>SHMALL</code>	0x200000	<code>shmall</code>	Numero massimo di pagine che possono essere usate per i segmenti di memoria condivisa.
<code>SHMMAX</code>	0x2000000	<code>shmmmax</code>	Dimensione massima di un segmento di memoria condivisa.
<code>SHMMNI</code>	4096	<code>msgmni</code>	Numero massimo di segmenti di memoria condivisa presenti nel kernel.
<code>SHMMIN</code>	1	—	Dimensione minima di un segmento di memoria condivisa.
<code>SHMLBA</code>	<code>PAGE_SIZE</code>	—	Limite inferiore per le dimensioni minime di un segmento (deve essere allineato alle dimensioni di una pagina di memoria).
<code>SHMSEG</code>	—	—	Numero massimo di segmenti di memoria condivisa per ciascun processo.

**Tabella 12.4:** 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 che permette di effettuare le operazioni di controllo su un segmento di memoria condivisa è `shmctl`; il suo prototipo è:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
    Esegue le operazioni di controllo su un segmento di memoria condivisa.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

<b>EACCES</b>	Si è richiesto <code>IPC_STAT</code> ma i permessi non consentono l'accesso in lettura al segmento.
<b>EINVAL</b>	O <code>shmid</code> non è un identificatore valido o <code>cmd</code> non è un comando valido.
<b>EIDRM</b>	L'argomento <code>shmid</code> fa riferimento ad un segmento che è stato cancellato.
<b>EPERM</b>	Si è specificato un comando con <code>IPC_SET</code> o <code>IPC_RMID</code> senza i permessi necessari.
<b>EOVERFLOW</b>	Si è tentato il comando <code>IPC_STAT</code> ma il valore del group-ID o dell'user-ID è troppo grande per essere memorizzato nella struttura puntata dal <code>buf</code> .
<b>EFAULT</b>	L'indirizzo specificato con <code>buf</code> non è valido.

Il comando specificato attraverso l'argomento `cmd` determina i diversi effetti della funzione; i possibili valori che esso può assumere, ed il corrispondente comportamento della funzione, sono i seguenti:

<b>IPC_STAT</b>	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.
<b>IPC_RMID</b>	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 user-ID effettivo corrispondente o al creatore del segmento, o al proprietario del segmento, o all'amministratore.
<b>IPC_SET</b>	Permette di modificare i permessi ed il proprietario del segmento. Per modificare i valori di <code>shm_perm.mode</code> , <code>shm_perm.uid</code> e <code>shm_perm.gid</code> occorre essere il proprietario o il creatore del segmento, oppure l'amministratore. Compiuta l'operazione aggiorna anche il valore del campo <code>shm_ctime</code> .
<b>SHM_LOCK</b>	Abilita il <i>memory locking</i> <sup>32</sup> sul segmento di memoria condivisa. Solo l'amministratore può utilizzare questo comando.
<b>SHM_UNLOCK</b>	Disabilita il <i>memory locking</i> sul segmento di memoria condivisa. Solo l'amministratore può utilizzare questo comando.

i primi tre comandi sono gli stessi già visti anche per le code di messaggi e gli insiemi di semafori, gli ultimi due sono delle estensioni specifiche previste da Linux, che permettono di abilitare e disabilitare il meccanismo della memoria virtuale per il segmento.

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

<sup>32</sup>impedisce cioè che la memoria usata per il segmento venga salvata su disco dal meccanismo della memoria virtuale; si ricordi quanto trattato in sez. 2.2.7.

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 al processo un segmento di memoria condivisa.

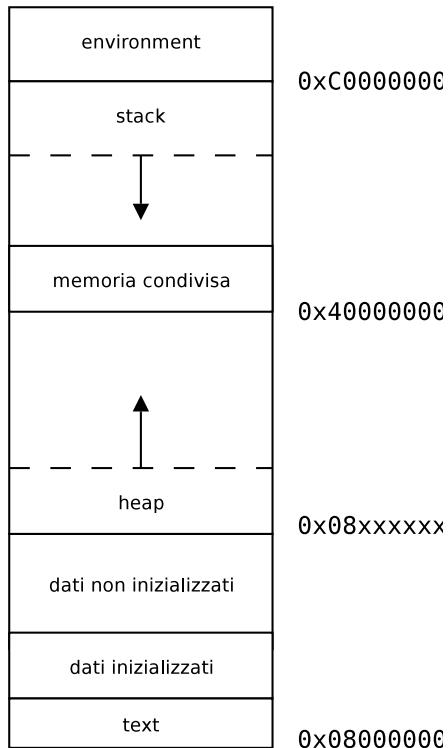
La funzione restituisce l'indirizzo del segmento in caso di successo, e -1 in caso di errore, nel qual caso **errno** assumerà i valori:

**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**.

ed inoltre **ENOMEM**.

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. 12.23 (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. 2.2.6) non viene influenzato. Si tenga presente infine che la funzione ha successo anche se il segmento è stato marcato per la cancellazione.



**Figura 12.23:** Disposizione dei segmenti di memoria di un processo quando si è agganciato un segmento di memoria condivisa.

L'argomento **shmaddr** specifica a quale indirizzo<sup>33</sup> 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

<sup>33</sup>Lo standard SVID prevede che l'argomento **shmaddr** sia di tipo **char \***, così come il valore di ritorno della funzione. In Linux è stato così con le *libc4* e le *libc5*, con il passaggio alle *glibc* il tipo di **shmaddr** è divenuto un **const void \*** e quello del valore di ritorno un **void \***.

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 sono solo due e sono identificati dalle costanti `SHM_RND` e `SHM_RDONLY`, 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, in Linux è equivalente a `PAGE_SIZE`).

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.

In caso di successo la funzione aggiorna anche i seguenti campi di `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. 3.2.2 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 sganciati. Lo stesso avviene all'uscita del processo attraverso una `exit`.

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 restituisce 0 in caso di successo, e -1 in caso di 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.

---

```

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 12.24:** Il codice delle funzioni che permettono di creare, trovare e rimuovere un segmento di memoria condivisa.

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. 12.24.

La prima funzione (3-16) 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 possibile quando la comunicazione non è sequenziale<sup>34</sup> 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. 12.25 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

---

<sup>34</sup>come accennato in sez. 12.2.4 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.

---

```

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 remaining 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("~/gapil/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         DirScan(argv[1], ComputeValues); /* execute scan */
46         MutexUnlock(mutex); /* unlock shared memory */
47         sleep(pause); /* sleep until next watch */
48     }
49 }
```

---

**Figura 12.25:** Codice della funzione principale del programma `DirMonitor.c`.

(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 il parametro necessario contenente il nome della directory da tenere sotto controllo, senza il quale esce immediatamente con un messaggio di errore.

Poi, per verificare che il parametro 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`.<sup>35</sup> 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,<sup>36</sup> 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 descritte in sez. 12.2.5, anche un mutex, che utilizzeremo per regolare l'accesso alla memoria condivisa.

---

```

1 /* Routine to compute directory properties inside DirScan */
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 12.26:* Codice delle funzioni ausiliarie usate da `DirMonitor.c`.

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 (42-48) all'interno di un ciclo infinito: 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

<sup>35</sup>si noti come si è potuta fare questa scelta, nonostante le indicazioni illustrate in sez. 10.1.5, per il particolare scopo del programma, che necessita comunque di restare all'interno di una directory.

<sup>36</sup>si 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.

utilizzando la funzione `DirScan`; 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` con una `sleep`.

Si noti come per il calcolo dei valori da mantenere nella memoria condivisa si sia usata ancora una volta la funzione `DirScan`, già utilizzata (e descritta in dettaglio) in sez. 5.1.6, 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.

Il codice di quest'ultima è riportato in fig. 12.26. 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 `DirScan`, 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. 5.3, si contano (8-14) quanti ce ne sono per ciascun tipo.

In fig. 12.26 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, per %d byte\n",
25           shmptr->tot_files, shmptr->tot_size);
26    MutexUnlock(mutex); /* unlock shared memory */
27 }
```

---

**Figura 12.27:** 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. 12.27. 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
0xffffffff 54067205  piccardi  666          4096          1

----- Semaphore Arrays -----
key      semid      owner      perms      nsems
0xffffffff 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 rilettura 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
```

## 12.3 Tecniche alternative

Come abbiamo detto in sez. 12.2.1, e ripreso nella descrizione dei singoli oggetti che ne fanno parte, il *SysV IPC* presenta numerosi problemi; in [1]<sup>37</sup> 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.

### 12.3.1 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. 12.1.5) 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.

### 12.3.2 I file di lock

Come illustrato in sez. 12.2.5 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.

---

<sup>37</sup>in particolare nel capitolo 14.

La prima possibilità, utilizzata fin dalle origini di Unix, è quella di usare dei *file di lock* (per i quali esiste anche una opportuna directory, `/var/lock`, nel filesystem standard). Per questo si usa la caratteristica della funzione `open` (illustrata in sez. 6.2.1) che prevede<sup>38</sup> 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. 12.28 (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`.

---

```

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 12.28:** Il codice delle funzioni `LockFile` e `UnlockFile` che permettono di creare e rimuovere un *file di lock*.

Uno dei limiti di questa tecnica è che, come abbiamo già accennato in sez. 6.2.1, 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 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.

Un 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

<sup>38</sup>questo 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.

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.

### 12.3.3 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. 11.4) 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.

Il codice delle varie funzioni usate per implementare un mutex utilizzando il file locking è riportato in fig. 12.29; 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. 11.4.3) 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)

---

```

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. */
12int LockMutex(int fd)
13{
14    struct flock lock;           /* file lock structure */
15    /* set flock structure */
16    lock.l_type = F_WRLCK;      /* set type: read or write */
17    lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
18    lock.l_start = 0;           /* set the start of the locked region */
19    lock.l_len = 0;             /* set the length of the locked region */
20    /* do locking */
21    return fcntl(fd, F_SETLKW, &lock);
22}
23/* Function UnlockMutex: unlock a file. */
24int UnlockMutex(int fd)
25{
26    struct flock lock;           /* file lock structure */
27    /* set flock structure */
28    lock.l_type = F_UNLCK;      /* set type: unlock */
29    lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
30    lock.l_start = 0;           /* set the start of the locked region */
31    lock.l_len = 0;             /* set the length of the locked region */
32    /* do locking */
33    return fcntl(fd, F_SETLK, &lock);
34}
35/* Function RemoveMutex: remove a mutex (unlinking the lock file). */
36int RemoveMutex(const char *path_name)
37{
38    return unlink(path_name);
39}
40/* Function ReadMutex: read a mutex status. */
41int ReadMutex(int fd)
42{
43    int res;
44    struct flock lock;           /* file lock structure */
45    /* set flock structure */
46    lock.l_type = F_WRLCK;      /* set type: unlock */
47    lock.l_whence = SEEK_SET;   /* start from the beginning of the file */
48    lock.l_start = 0;           /* set the start of the locked region */
49    lock.l_len = 0;             /* set the length of the locked region */
50    /* do locking */
51    if ( (res = fcntl(fd, F_GETLK, &lock)) ) {
52        return res;
53    }
54    return lock.l_type;
55}

```

---

**Figura 12.29:** Il codice delle funzioni che permettono per la gestione dei *mutex* con il file locking.

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`<sup>39</sup> 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. 11.4.3; 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.

### 12.3.4 Il *memory mapping* anonimo

Abbiamo già visto che quando i processi sono correlati<sup>40</sup> l’uso delle pipe può costituire una valida alternativa alle code di messaggi; nella stessa situazione si può evitare l’uso di una memoria condivisa facendo ricorso al cosiddetto *memory mapping* anonimo.

In sez. 11.3.2 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.<sup>41</sup> Vedremo come utilizzare questa tecnica più avanti, quando realizzeremo una nuova versione del monitor visto in sez. 12.2.6 che possa restituiscia i risultati via rete.

## 12.4 Il sistema di comunicazione fra processi di POSIX

Per superare i numerosi problemi del *SysV IPC*, evidenziati per i suoi aspetti generali in coda a sez. 12.2.1 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.

### 12.4.1 Considerazioni generali

In Linux non tutti gli oggetti del *POSIX IPC* sono pienamente supportati nel kernel ufficiale; solo la memoria condivisa è presente con l’interfaccia completa, ma solo a partire dal kernel 2.4.x, i semafori sono forniti dalle *glibc* nella sezione che implementa i thread POSIX, le code di messaggi non hanno alcun tipo di supporto ufficiale. Per queste ultime esistono tuttavia dei patch e una libreria aggiuntiva.

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

---

<sup>39</sup>non 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.

<sup>40</sup>se cioè hanno almeno un progenitore comune.

<sup>41</sup>nei 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 riletti secondo le stesse modalità usate nel *memory mapping* anonimo.

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 pertanto subordinato in maniera quasi completa alla relativa implementazione.<sup>42</sup> Nel caso di Linux, sia per quanto riguarda la memoria condivisa, che per quanto riguarda 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. 12.4.4 e sez. 12.4.2) ed i nomi specificati nelle relative funzioni sono considerati come un *pathname* assoluto (comprendente eventuali sottodirectory) rispetto a queste radici.

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,<sup>43</sup> che funzionano come su dei file normali.

In particolare i permessi associati agli oggetti di IPC POSIX sono identici ai permessi dei file, e il controllo di accesso segue esattamente la stessa semantica (quella illustrata in sez. 5.3), invece di quella particolare (si ricordi quanto visto in sez. 12.2.2) 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 (essi corrispondono cioè a user id e group id effettivi del processo che esegue la creazione).

### 12.4.2 Code di messaggi

Le code di messaggi non sono ancora supportate nel kernel ufficiale, esiste però una implementazione sperimentale di Michal Wronski e Krzysztof Benedyczak,<sup>44</sup>. 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 e memoria condivisa con tutta la flessibilità che occorre.

Per poter utilizzare le code di messaggi, oltre ad utilizzare un kernel cui siano stati opportunamente applicati i relativi patch, occorre utilizzare la libreria mqueue<sup>45</sup> che contiene le funzioni dell'interfaccia POSIX.<sup>46</sup>

La libreria inoltre richiede la presenza dell'apposito filesystem di tipo mqueue montato su /dev/mqueue; questo può essere fatto aggiungendo ad /etc/fstab una riga come:

---

<sup>42</sup>tanto che Stevens in [11] cita questo caso come un esempio della maniera standard usata dallo standard POSIX per consentire implementazioni non standardizzabili.

<sup>43</sup>questo è ancora più vero nel caso di Linux, che usa una implementazione che lo consente, non è detto che altrettanto valga per altri kernel. In particolare sia la memoria condivisa che per le code di messaggi, come si può facilmente evincere con uno strace, le system call utilizzate sono le stesse, in quanto esse sono realizzate con dei file in speciali filesystem.

<sup>44</sup>i patch al kernel e la relativa libreria possono essere trovati su [http://www.mat.uni.torun.pl/~wrona posix\\_ipc](http://www.mat.uni.torun.pl/~wrona posix_ipc), questi sono stati inseriti nel kernel ufficiale a partire dalla versione 2.6.6-rc1.

<sup>45</sup>i programmi che usano le code di messaggi cioè devono essere compilati aggiungendo l'opzione -lmqueue al comando gcc, dato che le funzioni non fanno parte della libreria standard, in corrispondenza all'inclusione del supporto nel kernel ufficiale, anche le relative funzioni sono state inserite nelle glibc a partire dalla versione 2.3.4.

<sup>46</sup>in realtà l'implementazione è realizzata tramite delle speciali chiamate ad ioctl sui file del filesystem speciale su cui vengono mantenuti questi oggetti di IPC.

<code>mqueue</code>	<code>/dev/mqueue</code>	<code>mqueue</code>	<code>defaults</code>	0	0
---------------------	--------------------------	---------------------	-----------------------	---	---

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 che permette di aprire (e crearla se non esiste ancora) una coda di messaggi POSIX è `mq_open`, ed il suo prototipo è:

```
#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 restituisce il descrittore associato alla coda in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori:

<code>EACCES</code>	Il processo non ha i privilegi per accedere al alla memoria secondo quanto specificato da <code>oflag</code> .
<code>EEXIST</code>	Si è specificato <code>O_CREAT</code> e <code>O_EXCL</code> ma la coda già esiste.
<code>EINTR</code>	La funzione è stata interrotta da un segnale.
<code>EINVAL</code>	Il file non supporta la funzione, o si è specificato <code>O_CREAT</code> con una valore non nullo di <code>attr</code> e valori non validi di <code>mq_maxmsg</code> e <code>mq_msgsize</code> .
<code>ENOENT</code>	Non si è specificato <code>O_CREAT</code> ma la coda non esiste.

ed inoltre `ENOMEM`, `ENOSPC`, `EFAULT`, `EMFILE` ed `ENFILE`.

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`.<sup>47</sup> Se la coda esiste già il descrittore farà riferimento allo stesso oggetto, consentendo così la comunicazione fra due processi diversi.

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 tab. 6.2 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

<sup>47</sup>nella implementazione citata questo è definito come `int`.

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`; i valori di quest'ultimo sono identici a quelli usati per `open`, 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. 12.30.

---

```
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 queued */
};
```

---

**Figura 12.30:** 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_msgsize` e `mq_maxmsg`, che indicano rispettivamente la dimensione massima di un messaggio ed il numero massimo di messaggi che essa può contenere. Il valore dovrà essere positivo e minore dei rispettivi limiti di sistema `MQ_MAXMSG` e `MQ_MSGSIZE`, altrimenti la funzione fallirà con un errore di `EINVAL`. Qualora si specifichi per `attr` un puntatore nullo gli attributi della coda saranno impostati ai valori predefiniti.

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 la coda mqdes.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori `EBADF` o `EINTR`.

La funzione è analoga a `close`,<sup>48</sup> 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 `mq_unlink`, il cui prototipo è:

```
#include <mqqueue.h>
int mq_unlink(const char *name)
    Rimuove una coda di messaggi.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà gli stessi valori riportati da `unlink`.

Anche in questo caso il comportamento della funzione è analogo a quello di `unlink` per i file,<sup>49</sup> la funzione remove la coda `name`, così che una successiva chiamata a `mq_open` fallisce o crea una coda diversa.

<sup>48</sup>in Linux, dove le code sono implementate come file su un filesystem dedicato, è esattamente la stessa funzione.

<sup>49</sup>di nuovo l'implementazione di Linux usa direttamente `unlink`.

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 in precedenza 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:

```
#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 restituiscono 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori `EBADF` o `EINVAL`.

La funzione `mq_getattr` legge i valori correnti degli attributi della coda nella struttura 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.

La funzione `mq_setattr` permette di modificare gli attributi di una coda tramite i valori contenuti nella struttura puntata da `mqstat`, ma può essere modificato solo il campo `mq_flags`, gli altri campi vengono 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, `mq_send` e `mq_timedsend`, i cui 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.
int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned
    msg_prio, const struct timespec *abs_timeout)
    Esegue l'inserimento di un messaggio su una coda entro il tempo abs_timeout.
```

Le funzioni restituiscono 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori:

<code>EAGAIN</code>	Si è aperta la coda con <code>O_NONBLOCK</code> , e la coda è piena.
<code>EMSGSIZE</code>	La lunghezza del messaggio <code>msg_len</code> eccede il limite impostato per la coda.
<code>ENOMEM</code>	Il kernel non ha memoria sufficiente. Questo errore può avvenire quando l'inserimento del messaggio
<code>EINVAL</code>	Si è specificato un valore nullo per <code>msg_len</code> , o un valore di <code>msg_prio</code> fuori dai limiti, o un valore non valido per <code>abs_timeout</code> .
<code>ETIMEDOUT</code>	L'inserimento del messaggio non è stato effettuato entro il tempo stabilito. ed inoltre <code>EBADF</code> ed <code>EINTR</code> .

Entrambe le funzioni richiedono un puntatore al testo del messaggio 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; i messaggi di priorità maggiore vengono inseriti davanti a quelli di priorità inferiore (e quindi saranno riletti per primi). A parità del valore della priorità il messaggio sarà inserito in coda a tutti quelli con la stessa priorità. Il valore della priorità non può eccedere il limite di sistema `MQ_PRIO_MAX`, che nel caso 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 la modalità non bloccante, nel qual caso entrambe ritornano `EAGAIN`. La sola differenza fra le due funzioni è che la seconda, passato il tempo massimo impostato con l'argomento `abs_timeout`, ritorna comunque con un errore di `ETIMEDOUT`.

Come per l'inserimento, anche per l'estrazione dei messaggi da una coda sono previste due funzioni, `mq_receive` e `mq_timedreceive`, i cui prototipi sono:

```
#include <mqqueue.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.
ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int
                        *msg_prio, const struct timespec *abs_timeout)
    Effettua la ricezione di un messaggio da una coda entro il tempo abs_timeout.
```

Le funzioni restituiscono il numero di byte del messaggio in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori:

<code>EAGAIN</code>	Si è aperta la coda con <code>O_NONBLOCK</code> , e la coda è vuota.
<code>EMSGSIZE</code>	La lunghezza del messaggio sulla coda eccede il valore <code>msg_len</code> specificato per la ricezione.
<code>EINVAL</code>	Si è specificato un valore nullo per <code>msg_ptr</code> , o un valore non valido per <code>abs_timeout</code> .
<code>ETIMEDOUT</code>	La ricezione del messaggio non è stata effettuata entro il tempo stabilito. ed inoltre <code>EBADF</code> , <code>EINTR</code> , <code>ENOMEM</code> , o <code>EINVAL</code> .

La funzione estrae dalla coda 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.

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_getaddr` 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`.

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 non interessi usare la priorità dei messaggi si

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 [11], 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 <mqueue.h>
int mq_notify(mqd_t mqdes, const struct sigevent *notification)
    Attiva il meccanismo di notifica per la coda mqdes.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà i valori:

<code>EBUSY</code>	C'è già un processo registrato per la notifica.
<code>EBADF</code>	Il descrittore non fa riferimento ad una coda di messaggi.

Il meccanismo di notifica permette di segnalare in maniera asincrona ad un processo la presenza di dati sulla coda, 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 dal valore dell'argomento `notification`, che è un puntatore ad una apposita struttura `sigevent`, (definita in fig. 11.3) introdotta dallo standard POSIX.1b per gestire la notifica di eventi; per altri dettagli si può vedere quanto detto in sez. 11.2.2 a proposito della stessa struttura per l'invio dei segnali usati per l'I/O asincrono.

Attraverso questa struttura si possono impostare le modalità con cui viene effettuata la notifica; in particolare il campo `sigev_notify` deve essere posto a `SIGEV_SIGNAL`<sup>50</sup> ed il campo `sigev_signo` deve indicare il valore del segnale che sarà inviato al processo. Inoltre il campo `sigev_value` è il puntatore ad una struttura `sigval_t` (definita in fig. 9.13) che permette di restituire al gestore del segnale un valore numerico o un indirizzo,<sup>51</sup> posto che questo sia installato nella forma estesa vista in sez. 9.4.3.

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 `NULL`. Dato che un solo processo alla volta può essere registrato, la funzione fallisce con `EBUSY` se c'è un altro processo già registrato. 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

<sup>50</sup>il meccanismo di notifica basato sui thread, specificato tramite il valore `SIGEV_THREAD`, non è implementato.

<sup>51</sup>per il suo uso si riveda la trattazione fatta in sez. 9.4.6 a proposito dei segnali real-time.

segnale di notifica. A differenza della situazione simile che si aveva con i segnali non affidabili,<sup>52</sup> 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. 9.9). In particolare `si_pid` viene impostato al valore del `pid` del processo che ha emesso il segnale, `si_uid` all'userid effettivo, `si_code` a `SI_MESSAGE`, 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<sup>53</sup>

### 12.4.3 Semafori

Dei semafori POSIX esistono sostanzialmente due implementazioni; una è fatta a livello di libreria ed è fornita dalla libreria dei thread; questa però li implementa solo a livello di thread e non di processi.<sup>54</sup> Esiste però anche una libreria realizzata da Konstantin Knizhnik, che reimplementa l'interfaccia POSIX usando i semafori di SysV IPC, e che non vale comunque la pena di usare visto che i problemi sottolineati in sez. 12.2.5 rimangono, anche se mascherati.

In realtà a partire dal kernel 2.5.7 è stato introdotto un meccanismo di sincronizzazione completamente nuovo, basato sui cosiddetti *futexes*<sup>55</sup>, con il quale dovrebbe essere possibile implementare una versione nativa dei semafori; esso è già stato usato con successo per reimplementare in maniera più efficiente tutte le direttive di sincronizzazione previste per i thread POSIX. L'interfaccia corrente è stata stabilizzata a partire dal kernel 2.5.40.

### 12.4.4 Memoria condivisa

La memoria condivisa è l'unico degli oggetti di IPC POSIX già presente nel kernel ufficiale; in realtà il supporto a questo tipo di oggetti è realizzato attraverso il filesystem `tmpfs`, uno speciale filesystem che mantiene tutti i suoi contenuti in memoria,<sup>56</sup> che viene attivato abilitando l'opzione `CONFIG_TMPFS` in fase di compilazione del kernel.

Per potere utilizzare l'interfaccia POSIX per le code di messaggi le *glibc*<sup>57</sup> richiedono di compilare i programmi con l'opzione `-lrt`; inoltre è necessario che in `/dev/shm` sia montato un filesystem `tmpfs`; questo di norma viene eseguita aggiungendo una riga tipo:

```
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
```

---

<sup>52</sup>l'argomento è stato affrontato in 9.1.2.

<sup>53</sup>di nuovo si faccia riferimento a quanto detto al proposito in sez. 9.4.3 e sez. 9.4.6.

<sup>54</sup>questo significa che i semafori sono visibili solo all'interno dei thread creati da un singolo processo, e non possono essere usati come meccanismo di sincronizzazione fra processi diversi.

<sup>55</sup>la sigla sta per *fast user mode mutex*.

<sup>56</sup>il filesystem `tmpfs` è diverso da un normale RAM disk, anch'esso disponibile attraverso il filesystem `ramfs`, proprio perché realizza una interfaccia utilizzabile anche per la memoria condivisa; esso infatti non ha dimensione fissa, ed usa direttamente la cache interna del kernel (che viene usata anche per la shared memory in stile SysV). In più i suoi contenuti, essendo trattati direttamente dalla memoria virtuale possono essere salvati sullo swap automaticamente.

<sup>57</sup>le funzioni sono state introdotte con le glibc-2.2.

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 <mqqueue.h>
int shm_open(const char *name, int oflag, mode_t mode)
    Apre un segmento di memoria condivisa.
```

La funzione restituisce un file descriptor positivo in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà gli stessi valori riportati da `open`.

La funzione apre un segmento di memoria condivisa identificato dal nome `name`. Come già spiegato in sez. 12.4.1 questo nome può essere specificato in forma standard solo facendolo iniziare per / e senza ulteriori /, Linux supporta comunque nomi generici, che verranno intepretati prendendo come radice `/dev/shm`.<sup>58</sup>

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 tab. 6.2 dei quali però `shm_open` riconosce solo i seguenti:

<code>O_RDONLY</code>	Apre il file descriptor associato al segmento di memoria condivisa per l'accesso in sola lettura.
<code>O_RDWR</code>	Apre il file descriptor associato al segmento di memoria condivisa per l'accesso in lettura e scrittura.
<code>O_CREAT</code>	Necessario qualora si debba creare il segmento di memoria condivisa se esso non esiste; in questo caso viene usato il valore di <code>mode</code> per impostare i permessi, che devono essere compatibili con le modalità con cui si è aperto il file.
<code>O_EXCL</code>	Se usato insieme a <code>O_CREAT</code> fa fallire la chiamata a <code>shm_open</code> se il segmento esiste già, altrimenti esegue la creazione atomicamente.
<code>O_TRUNC</code>	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`<sup>59</sup> viste in sez. 6.2.1; in particolare viene 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 di dati, 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 il segmento può essere creato specificando `O_CREAT`; in tal caso il segmento avrà (così come i nuovi file) lunghezza nulla. Dato che un segmento di lunghezza nulla è di scarsa utilità, per impostarne la dimensione si deve usare `ftruncate` (vedi sez. 5.2.3), prima di mapparlo in memoria con `mmap`. Si tenga presente che una volta chiamata `mmap` si può chiudere il file descriptor (con `close`), senza che la mappatura ne risenta.

Come per i file, quando si vuole effettivamente rimuovere segmento di memoria condivisa, occorre usare la funzione `shm_unlink`, il cui prototipo è:

<sup>58</sup>occorre pertanto evitare di specificare qualcosa del tipo `/dev/shm/nome` all'interno di `name`, perché questo comporta, da parte delle routine di libereria, il tentativo di accedere a `/dev/shm/dev/shm/nome`.

<sup>59</sup>in realtà, come accennato, `shm_open` è un semplice wrapper per `open`, usare direttamente quest'ultima avrebbe lo stesso effetto.

```
#include <mqueue.h>
int shm_unlink(const char *name)
    Rimuove un segmento di memoria condivisa.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore; nel quel caso `errno` assumerà gli stessi valori riportati da `unlink`.

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.

Come esempio per l'uso di queste funzioni vediamo come è possibile riscrivere una interfaccia semplificata analoga a quella vista in fig. 12.24 per la memoria condivisa in stile SysV. Il codice, riportato in fig. 12.31, è sempre 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 già 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 puntatore 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.

---

```

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 12.31:** Il codice delle funzioni di gestione dei segmenti di memoria condivisa POSIX.



## **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 al modello più comune usato 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 protocollo 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, come gli esempi che abbiamo usato in cap. 12 a proposito della comunicazione fra processi nello stesso sistema, non fanno necessariamente uso della rete.

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 per lo stesso Napster, in cui le ricerche venivano 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 stato 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.

## 13.2 I protocollli 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 una serie di protocolli, il più famoso dei quali, quello alla base del funzionamento di internet, è il protocollo 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>	Applicazione
Livello 6	<i>Presentation</i>	Presentazione
Livello 5	<i>Session</i>	Sessione
Livello 4	<i>Transport</i>	Trasporto
Livello 3	<i>Network</i>	Rete
Livello 2	<i>DataLink</i>	Collegamento Dati
Livello 1	<i>Physical</i>	Connessione Fisica

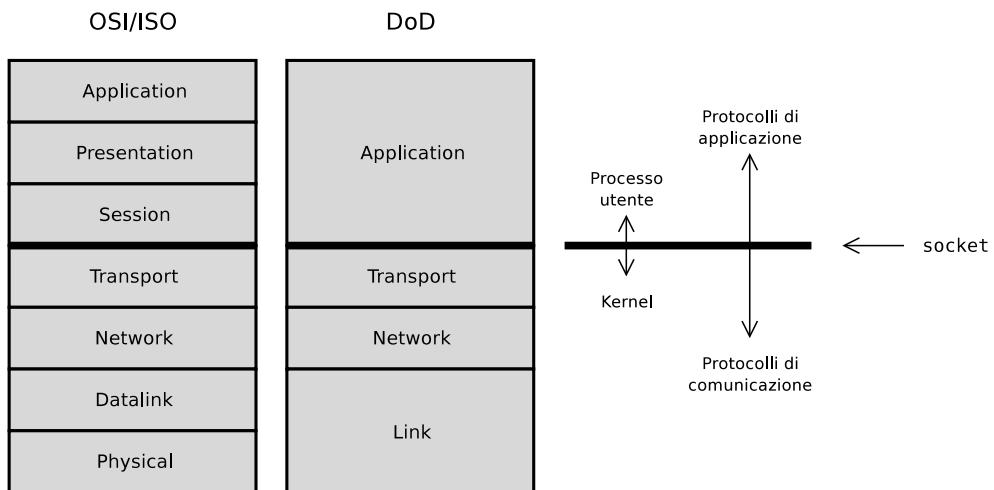
**Tabella 13.1:** I sette livelli del protocollo ISO/OSI.

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 complesso<sup>1</sup>, 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.

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,

<sup>1</sup>infatti 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.



**Figura 13.1:** Struttura a livelli dei protocolli OSI e TCP/IP, con la relative corrispondenze e la divisione fra kernel e user space.

mentre il modello TCP/IP è più vicino alla separazione concreta dei vari strati del sistema operativo; useremo pertanto quest'ultimo, anche per la sua maggiore semplicità.<sup>2</sup>

### 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.<sup>3</sup>

Livello	Nome	Esempi
Livello 4	<i>Application</i>	Applicazione
Livello 3	<i>Transport</i>	Trasporto
Livello 2	<i>Network</i>	Rete
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** È 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

<sup>2</sup>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*.

<sup>3</sup>in realtà è sempre possibile accedere dallo user space, attraverso una opportuna interfaccia (come vedremo in sez. 14.3.6), ai livelli inferiori del protocollo.

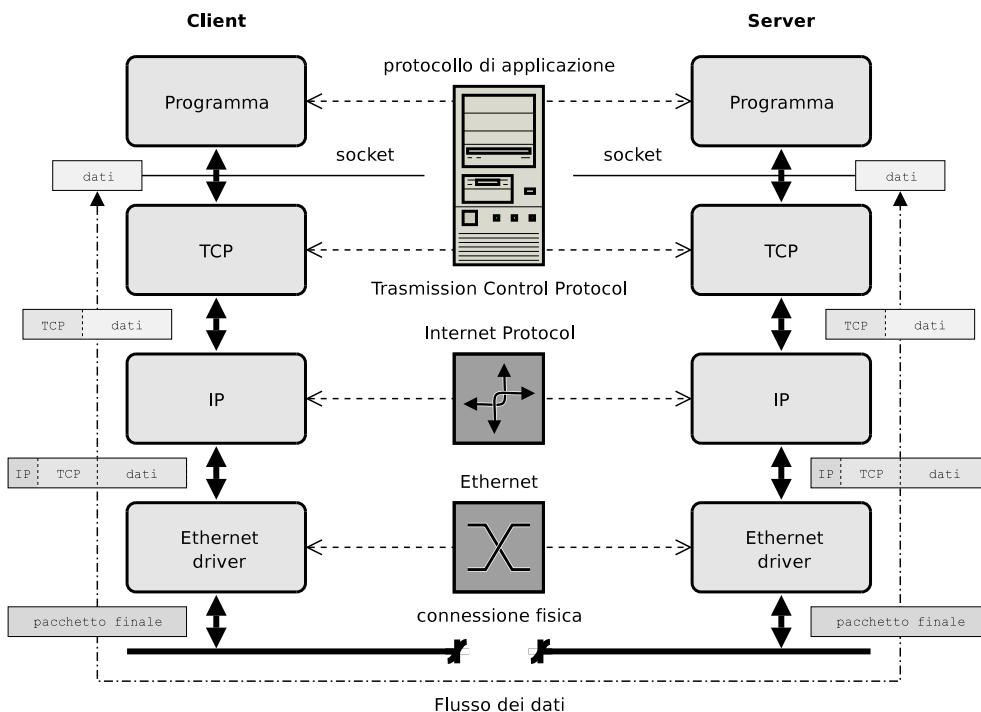
con recupero degli errori o inaffidabile. I protocolli principali di questo livello sono il TCP e l'UDP.

**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.



**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<sup>4</sup>).

<sup>4</sup>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 architettonica 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 allo livello di collegamento, 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 necessiti) 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* (*intradicatori*), 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 Il protocollo TCP/IP

Come accennato in sez. 13.2 il protocollo 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, 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

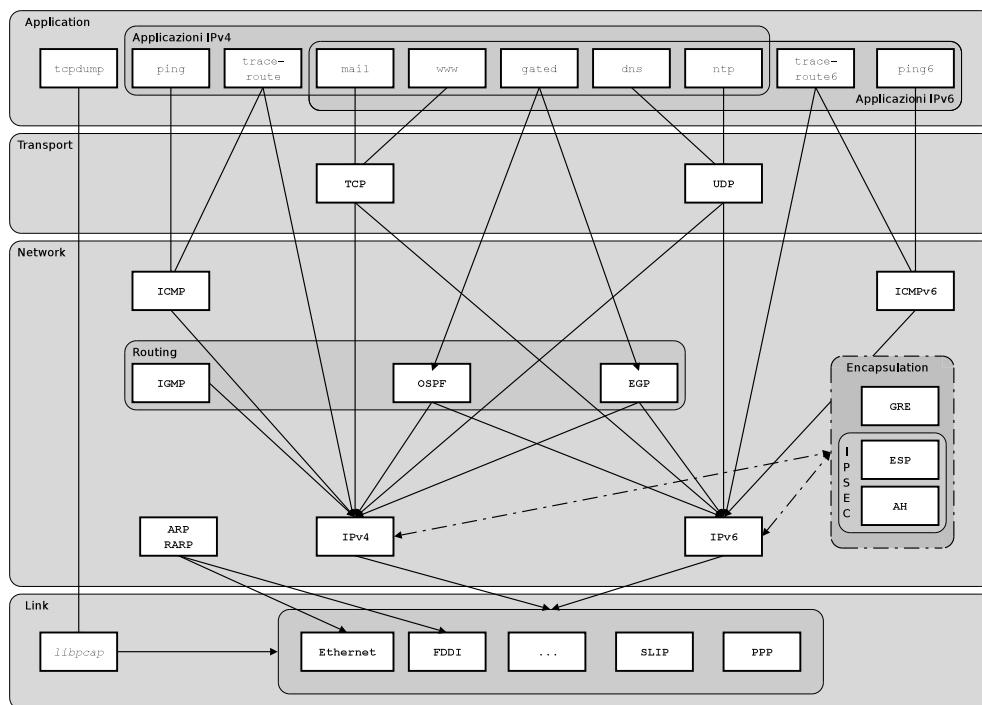
interfaccia di programmazione su questo confine, tanto più che è proprio lì (come evidenziato in fig. 13.1) che nei sistemi Unix (e non solo) viene inserita la divisione fra kernel space e user space.

In realtà in un sistema Unix è possibile accedere anche agli altri livelli inferiori (e non solo a quello di trasporto) con opportune interfacce di programmazione (vedi sez. 14.3.6), 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 dalle 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** *Internet Protocol version 4.* 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** *Internet Protocol version 6.* 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'autoconfigurazione, ed un nuovo tipo di indirizzi, gli *anycast*, 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, come l'acknowledgment, i timeout, la ritrasmissione, etc. 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 stradatori (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 *multicasting* (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.
- EIGP** *Exterior Gateway Protocol.* È un protocollo di routing usato per comunicare lo stato fra gateway vicini a livello di *sistemi autonomi*<sup>5</sup>, 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 un 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 encapsulamento che permette di encapsulare un qualunque altro protocollo all'interno di IP.
- AH** *Authentication Header.* 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.

---

<sup>5</sup>vengono chiamati *autonomous systems* i raggruppamenti al livello più alto della rete.

- ESP** *Encapsulating Security Payload.* 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.
- PPP** *Point-to-Point Protocol.* 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.
- SLIP** *Serial Line over IP.* 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 IP ci si riferisce in genere alla versione attualmente in uso che è la versione 4 (e viene pertanto chiamato IPv4). Questa versione venne standardizzata nel 1981 dall'RFC 719.

Internet Protocol 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, etc.). 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 IPv4 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, mantendone 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 autoconfigurazione 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 riprocessamento 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. A.1.

### 13.3.3 User Datagram Protocol (UDP)

UDP è un protocollo di trasporto molto semplice, la sua descrizione completa è contenuta dell'RFC 768, ma in sostanza esso è una semplice interfaccia a IP dal livello di trasporto. Quando un'applicazione usa UDP essa scrive un pacchetto di dati (il cosiddetto *datagram* che da il nome al protocollo) su un socket, al pacchetto viene aggiunto un header molto semplice (per una descrizione più accurata vedi sez. B.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 multicasting.

### 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 fra 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 anche più avanti, 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. A.2).
- 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 un 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<sup>6)</sup> in blocchi più piccoli che possono essere trasmessi attraverso l'interfaccia.

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 reti diverse.

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), mentre i router IPv4 si. 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 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 TCP definisce una MSS (*Maximum Segment Size*) che annuncia all'altro capo della connessione la dimensione massima dimensione 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.

<sup>6)</sup>questo 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. A.2.

# Capitolo 14

## Introduzione ai 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 Una panoramica

Iniziamo con una 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.

#### 14.1.1 I *socket*

I *socket*<sup>1</sup> sono uno dei principali meccanismi di comunicazione utilizzato in ambito Unix, e li abbiamo brevemente incontrati in sez. 12.1.5, fra i vari meccanismi di intercominazione 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. 12.1.1) ma, a differenza di questa e degli altri meccanismi esaminati nel capitolo cap. 12, 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).

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.

#### 14.1.2 Concetti base

Per capire il funzionamento dei socket occorre avere presente il funzionamento dei protocolli di rete (vedi cap. 13), ma l'interfaccia è del tutto generale e benché le problematiche (e quindi le

---

<sup>1</sup>una traduzione letterale potrebbe essere *presa*, ma essendo universalmente noti come *socket* utilizzeremo sempre la parola inglese.

modalità di risolvere i problemi) siano diverse a seconda del tipo di protocollo di comunicazione usato, le funzioni da usare 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 dello *stile* di comunicazione usato. La scelta di questo stile va infatti ad incidere sulla semantica che verrà utilizzata a livello utente per gestire la comunicazione (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 stili 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 inviati in blocchi separati.

Un’altro esempio di stile concerne la possibilità che la comunicazione possa o meno perdere dati, possa o meno non rispettare l’ordine in cui essi non sono inviati, o inviare dei pacchetti più volte (come nel caso di TCP e UDP).

Un terzo esempio di stile di comunicazione concerne le modalità in cui essa avviene, 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 *broadcast* come per la radio, in cui i pacchetti vengono emessi su appositi “canali” dove chiunque si collega possa riceverli.

È chiaro che ciascuno di questi stili comporta una modalità diversa di gestire la comunicazione, ad esempio se è inaffidabile occorrerà essere in grado di gestire la perdita o il rimescolamento dei dati, se è a pacchetti questi dovranno essere opportunamente trattati, ecc.

## 14.2 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.

### 14.2.1 La funzione `socket`

La creazione di un socket avviene attraverso l’uso della funzione `socket`; essa restituisce un *file descriptor*<sup>2</sup> che serve come riferimento al socket; il suo prototipo è:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
Apre un socket.
```

La funzione restituisce un intero positivo in caso di successo, e -1 in caso di fallimento, nel qual caso la variabile `errno` assumerà i valori:

<b>EPROTONOSUPPORT</b>	Il tipo di socket o il protocollo scelto non sono supportati nel dominio.
<b>ENFILE</b>	Il kernel non ha memoria sufficiente a creare una nuova struttura per il socket.
<b>EMFILE</b>	Si è ecceduta la tabella dei file.
<b>EACCES</b>	Non si hanno privilegi per creare un socket nel dominio o con il protocollo specificato.
<b>EINVAL</b>	Protocollo sconosciuto o dominio non disponibile.
<b>ENOBUFS</b>	Non c’è sufficiente memoria per creare il socket (può essere anche <b>ENOMEM</b> ).
Inoltre, a seconda del protocollo usato, potranno essere generati altri errori, che sono riportati nelle relative pagine di manuale.	

La funzione ha tre argomenti, `domain` specifica il dominio del socket (definisce cioè, come vedremo in sez. 14.2.2, la famiglia di protocolli usata), `type` specifica il tipo di socket (definisce

<sup>2</sup>del tutto analogo a quelli che si ottengono per i file di dati e le pipe, descritti in sez. 6.1.1.

cioè, come vedremo in sez. 14.2.3, 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.

### 14.2.2 Il dominio, o *protocol family*

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 inizia con una costante che inizia per `PF_`, iniziali di *protocol family*, un 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*,<sup>3</sup> dato che identificano il formato degli indirizzi usati in quel dominio per identificare i capi della comunicazione.

Nome	Valore	Utilizzo	Man page
<code>PF_UNSPEC</code>	0	Non specificato	
<code>PF_LOCAL</code>	1	Local communication	<code>unix(7)</code>
<code>PF_UNIX, PF_FILE</code>	1		
<code>PF_INET</code>	2	IPv4 Internet protocols	<code>ip(7)</code>
<code>PF_AX25</code>	3	Amateur radio AX.25 protocol	
<code>PF_IPX</code>	4	IPX - Novell protocols	
<code>PF_APPLETALK</code>	5	Appletalk	<code>ddp(7)</code>
<code>PF_NETROM</code>	6	Amateur radio NetROM	
<code>PF_BRIDGE</code>	7	Multiprotocol bridge	
<code>PF_ATMPVC</code>	8	Access to raw ATM PVCs	
<code>PF_X25</code>	9	ITU-T X.25 / ISO-8208 protocol	<code>x25(7)</code>
<code>PF_INET6</code>	10	IPv6 Internet protocols	<code>ipv6(7)</code>
<code>PF_ROSE</code>	11	Amateur Radio X.25 PLP	
<code>PF_DECnet</code>	12	Reserved for DECnet project	
<code>PF_NETBEUI</code>	13	Reserved for 802.2LLC project	
<code>PF_SECURITY</code>	14	Security callback pseudo AF	
<code>PF_KEY</code>	15	PF_KEY key management API	
<code>PF_NETLINK</code>	16	Kernel user interface device	<code>netlink(7)</code>
<code>PF_PACKET</code>	17	Low level packet interface	<code>packet(7)</code>
<code>PF_ASH</code>	18	Ash	
<code>PF_ECONET</code>	19	Acorn Econet	
<code>PF_ATMSVC</code>	20	ATM SVCs	
<code>PF_SNA</code>	22	Linux SNA Project	
<code>PF_IRDA</code>	23	IRDA sockets	
<code>PF_PPPOX</code>	24	PPPoX sockets	
<code>PF_WANPIPE</code>	25	Wanpipe API sockets	
<code>PF_BLUETOOTH</code>	31	Bluetooth sockets	

**Tabella 14.1:** Famiglie di protocolli definiti in Linux.

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

<sup>3</sup> nome che invece il manuale delle *glibc* riserva a quello che noi abbiamo chiamato domini.

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.<sup>4</sup>

I domini (e i relativi nomi simbolici), così come i nomi delle famiglie di indirizzi, sono definiti dall'header `socket.h`. Un elenco delle famiglie di protocolli disponibili in Linux è riportato in tab. 14.1.<sup>5</sup>

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 user-ID effettivo uguale a zero) o dotati della capability `CAP_NET_RAW`.

#### 14.2.3 Il tipo, o stile

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* [3] chiama *styles*) identificati dalle seguenti costanti:

<code>SOCK_STREAM</code>	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 <i>stream</i> ).
<code>SOCK_DGRAM</code>	Viene usato per trasmettere pacchetti di dati ( <i>datagram</i> ) di lunghezza massima prefissata, indirizzati singolarmente. Non esiste una connessione e la trasmissione è effettuata in maniera non affidabile.
<code>SOCK_SEQPACKET</code>	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, ed devono essere letti integralmente da ciascuna chiamata a <code>read</code> .
<code>SOCK_RAW</code>	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.
<code>SOCK_RDM</code>	Provvede un canale di trasmissione di dati affidabile, ma in cui non è garantito l'ordine di arrivo dei pacchetti.
<code>SOCK_PACKET</code>	Obsoleto, non deve essere usato.

Si tenga presente 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.

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 non il protocollo non abbia un nome definito, mentre si sono lasciate vuote le caselle per le combinazioni non supportate.

<sup>4</sup>in 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.

<sup>5</sup>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 `PF_MAX` che indica il valore massimo associabile ad un dominio (nel caso il suo valore 32).

Famiglia	Tipo				
	SOCK_STREAM	SOCK_DGRAM	SOCK_RAW	SOCK_PACKET	SOCK_SEQPACKET
PF_UNIX	si	si			
PF_INET	TCP	UDP	IPv4		
PF_INET6	TCP	UDP	IPv6		
PF_IPX					
PF_NETLINK		si	si		
PF_X25					si
PF_AX25					
PF_ATMPVC					
PF_APPLETALK		si	si		
PF_PACKET		si	si		

**Tabella 14.2:** Combinazioni valide di dominio e tipo di protocollo per la funzione `socket`.

## 14.3 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.

### 14.3.1 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.

---

```
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`.

In alcuni sistemi la struttura è leggermente diversa e prevede un primo membro aggiuntivo

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.

`uint8_t sin_len` (come riportato da R. Stevens in [2]). 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.

### 14.3.2 La struttura degli indirizzi IPv4

I socket di tipo `PF_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. 14.2, 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 14.2:** La struttura degli indirizzi dei socket internet (IPv4) `sockaddr_in`.

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 non prevede numeri di porta, che sono utilizzati solo dai protocolli di livello superiore come TCP e UDP. Questa struttura però viene usata anche per i socket RAW che accedono direttamente al livello di IP, nel qual caso il numero della porta viene 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 user-ID 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 reincontreremo 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*, questo comporta la necessità di usare apposite funzioni di conversione per mantenere la portabilità del codice (vedi sez. 14.4 per i dettagli del problema e le relative soluzioni).

### 14.3.3 La struttura degli indirizzi IPv6

Essendo IPv6 un'estensione di IPv4, i socket di tipo `PF_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. 14.3.

---

```
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 14.3:** La struttura degli indirizzi dei socket IPv6 `sockaddr_in6`.

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. A.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. Infine il campo `sin6_scope_id` è un campo introdotto in Linux con il kernel 2.4, per gestire alcune operazioni riguardanti il multicasting. 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.

### 14.3.4 La struttura degli indirizzi locali

I socket di tipo `PF_UNIX` o `PF_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. 12.1.5). 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. 14.4.

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 un file (di tipo socket) nel filesystem o una stringa univoca (mantenuta in uno spazio di nomi astratto). Nel primo caso l'indirizzo viene specificato come una stringa (terminata da uno zero) corrispondente al *pathname* del file;

---

```
#define UNIX_PATH_MAX    108
struct sockaddr_un {
    sa_family_t   sun_family;           /* AF_UNIX */
    char         sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

---

**Figura 14.4:** La struttura degli indirizzi dei socket locali (detti anche *unix domain*) `sockaddr_un` definita in `sys/un.h`.

nel secondo invece `sun_path` inizia con uno zero e vengono usati come nome i restanti byte come stringa, senza terminazione.

### 14.3.5 La struttura degli indirizzi AppleTalk

I socket di tipo `PF_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. 14.5; la struttura viene dichiarata includendo il file `netatalk/at.h`.

---

```
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 14.5:** La struttura degli indirizzi dei socket AppleTalk `sockaddr_atalk`.

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. 14.4.1); esso è composto da un 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, 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.

### 14.3.6 La struttura degli indirizzi dei *packet socket*

I *packet socket*, identificati dal dominio PF\_PACKET, sono un’interfaccia specifica di Linux per inviare e ricevere pacchetti direttamente su un’interfaccia di rete, senza passare per le routine 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, 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 (user-ID 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 passaggio ad una interfaccia specifica occorre usare la funzione bind 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_pktype;       /* Packet type */
    unsigned char  sll_halen;        /* Length of address */
    unsigned char  sll_addr[8];      /* Physical layer address */
};


```

---

**Figura 14.6:** 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. 14.6; 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, 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_pktype` indica il tipo di pacchetto; entrambi vengono impostati alla ricezione di un pacchetto ed han senso solo in questo caso. In particolare `sll_pktype` 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.

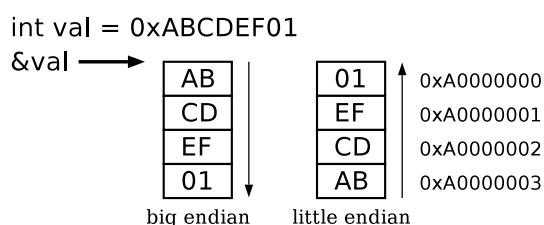
Si tenga presente infine che in fase di ricezione, anche se si richiede il troncamento del pacchetto, le funzioni `recvmsg`, `recv` e `recvfrom` restituiranno comunque la lunghezza effettiva del pacchetto così come arrivato sulla linea.

#### 14.4 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 in formato opportuno (*il network order*). Per capire cosa significa tutto ciò occorre introdurre un concetto generale che tornerà utile anche in seguito.

#### 14.4.1 La endianness

La rappresentazione di un numero binario in un computer 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 14.7:** Schema della disposizione dei dati in memoria a seconda della *endianess*.

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. 14.7 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*.

Si può allora verificare quale tipo di *endianess* 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 PC otterremo:

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

mentre su di un Mac avremo:

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

La *endianess* 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 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 resta sempre lo stesso, 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;                      /* endianess 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 14.8:** La funzione `endian`, usata per controllare il tipo di architettura della macchina.

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. 14.8, 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. 14.7, che sia *little endian*). Infine la funzione restituisce (12) il valore del confronto delle due variabili.

#### 14.4.2 Le funzioni per il riordinamento

Il problema connesso all'endianess 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 <netinet/in.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.

#### 14.4.3 Le funzioni `inet_aton`, `inet_addr` e `inet_ntoa`

Un secondo insieme di funzioni di manipolazione serve per passare dal formato binario usato nelle strutture degli indirizzi alla rappresentazione simbolica dei numeri IP che si usa normalmente.

Le prime tre funzioni di manipolazione riguardano la conversione degli indirizzi IPv4 da una stringa in cui il numero di IP è espresso secondo la cosiddetta notazione *dotted-decimal*, (cioè nella forma 192.168.0.1) al formato binario (direttamente in *network order*) e viceversa; in questo caso si usa la lettera `a` come mnemonico per indicare la stringa. Dette funzioni sono `inet_addr`, `inet_aton` e `inet_ntoa`, ed i rispettivi prototipi sono:

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *strptr)
    Converte la stringa dell'indirizzo dotted decimal in nel numero IP in network order.
int inet_aton(const char *src, struct in_addr *dest)
    Converte la stringa dell'indirizzo dotted decimal in un indirizzo IP.
char *inet_ntoa(struct in_addr addrptr)
    Converte un indirizzo IP in una stringa dotted decimal.

Tutte queste le funzioni non generano codice 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; per questo motivo essa è generalmente deprecata in favore di `inet_aton`.

La funzione `inet_aton` converte la stringa puntata da `src` nell'indirizzo binario che viene memorizzato nell'opportuna struttura `in_addr` (si veda fig. 14.2) 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 0 in caso di successo e 1 in caso di fallimento. Se usata con `dest` inizializzato a `NULL` effettua la validazione dell'indirizzo.

L'ultima funzione, `inet_ntoa`, converte il valore a 32 bit dell'indirizzo (espresso in *network order*) restituendo il puntatore alla stringa che contiene l'espressione in formato dotted decimal. Si deve tenere presente che la stringa risiede in memoria statica, per cui questa funzione non è rientrante.

#### 14.4.4 Le funzioni `inet_pton` e `inet_ntop`

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)
    Converte l'indirizzo espresso tramite una stringa nel valore numerico.
```

La funzione restituisce un valore negativo se `af` specifica una famiglia di indirizzi non valida, con `errno` che assume il valore `EAFNOSUPPORT`, un valore nullo se `src` non rappresenta un indirizzo valido, ed un valore positivo in caso di successo.

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 restituisce un valore positivo in caso di successo, nullo se la stringa non rappresenta un indirizzo valido, e negativo se `af` specifica una famiglia di indirizzi non valida.

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)
    Converte l'indirizzo dalla relativa struttura in una stringa simbolica.
```

La funzione restituisce un puntatore non nullo alla stringa convertita in caso di successo e `NULL` in caso di fallimento, nel qual caso `errno` assume i valori:

`ENOSPC` le dimensioni della stringa con la conversione dell'indirizzo eccedono la lunghezza specificata da `len`.

`ENOAFNOSUPPORT` 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 venire specificata attraverso 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.

Il formato usato per gli indirizzi in formato di presentazione è la notazione *dotted decimal* per IPv4 e quello descritto in sez. A.2.5 per IPv6.

# 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 stazioni, 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 è chiamato *three way handshake*; la successione tipica degli eventi (e dei segmenti<sup>1</sup> di dati che vengono scambiati) che porta alla creazione di una connessione è la seguente:

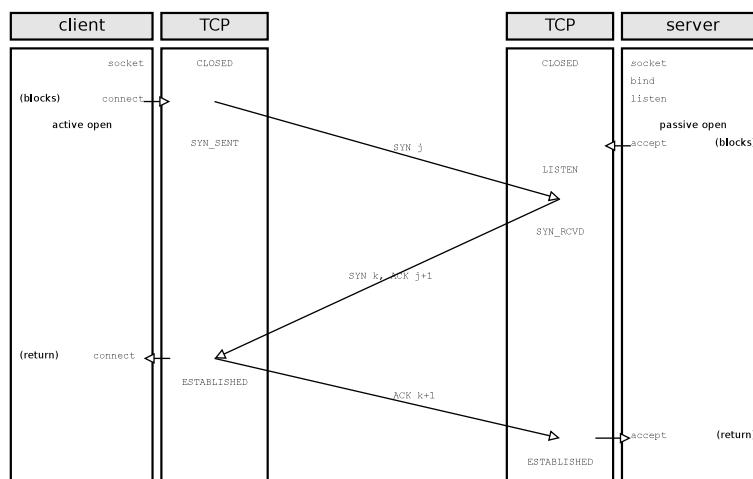
1. Il server deve essere preparato per accettare le connessioni in arrivo; il procedimento si chiama *apertura passiva* del socket (in inglese *passive open*). Questo viene fatto chiamando la sequenza di funzioni `socket`, `bind` e `listen`. Completata l'apertura passiva il server chiama la funzione `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 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 gli header IP e TCP (con il numero di sequenza iniziale e il flag SYN) e le opzioni di TCP.

---

<sup>1</sup>Si ricordi che il segmento è l'unità elementare di dati trasmessa dal protocollo TCP al livello successivo; tutti i segmenti hanno un header che contiene le informazioni che servono allo *stack TCP* (così viene di solito chiamata la parte del kernel che implementa il protocollo) per realizzare 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 vedere sez. B.1).

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'header 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'header 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 acknowledgement è sempre pari al numero di sequenza iniziale incrementato di uno; lo stesso varrà anche (vedi fig. 15.2) per l'acknowledgement di un FIN.

### 15.1.2 Le opzioni TCP.

Ciascun segmento SYN contiene in genere delle opzioni per il protocollo TCP (le cosiddette *TCP options*, che vengono inserite fra l'header e i dati) che servono a comunicare all'altro capo una serie di parametri utili a regolare la connessione. Normalmente vengono usate le seguenti opzioni:

- *MSS option*, dove MSS sta per *maximum segment size*, con questa opzione ciascun capo della connessione annuncia all'altro il massimo ammontare di dati che vorrebbe accettare per ciascun segmento nella connessione corrente. È possibile leggere e scrivere questo valore attraverso l'opzione del socket `TCP_MAXSEG`.

- *window scale option*, il protocollo TCP implementa il controllo di flusso attraverso una finestra annunciata (*advertized window*) 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 byte;<sup>2</sup> ma 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 questa opzione che indica un fattore di scala da applicare al valore della finestra annunciata<sup>3</sup> per la connessione corrente (espresso come numero di bit cui spostare a sinistra il valore della finestra annunciata inserito nel pacchetto).
- *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 come la precedente.

La MSS è generalmente supportata da quasi tutte le implementazioni 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 le opzioni.

### 15.1.3 La terminazione della connessione

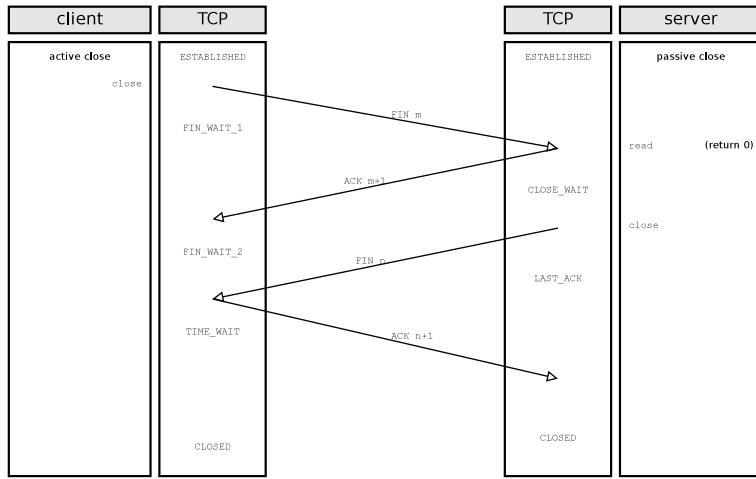
Mentre per la creazione di una connessione occorre un interscambio di tre segmenti, la procedura di chiusura ne richiede normalmente quattro. In questo caso la successione degli eventi è la seguente:

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 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 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 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. In fig. 15.2 si è rappresentato graficamente lo sequenza di scambio dei segmenti che conclude la connessione.

<sup>2</sup>Linux usa come massimo 32767 per evitare problemi con alcune implementazioni che usano l'aritmetica con segno per implementare lo stack TCP.

<sup>3</sup>essendo una nuova opzione per garantire la compatibilità con delle vecchie implementazioni 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.



**Figura 15.2:** La chiusura di una connessione TCP.

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.

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 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 anche se il caso più comune resta quello del client, ci sono alcuni servizi, il principale dei quali è l'HTTP, 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. B.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*.

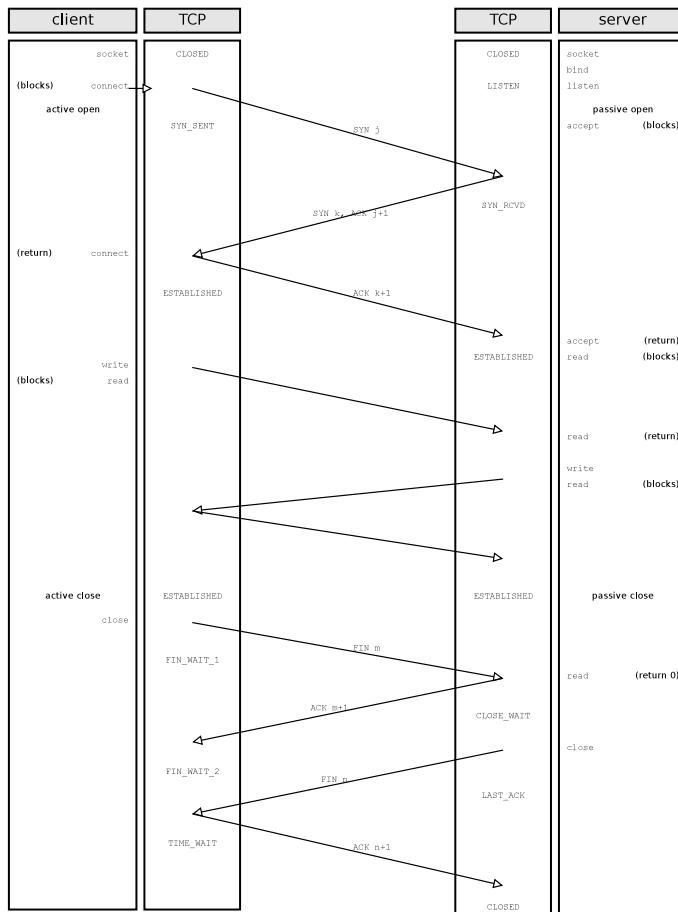
Non possiamo affrontare qui una descrizione completa del funzionamento del protocollo; un approfondimento sugli aspetti principali si trova in sez. B.1, ma per una trattazione completa il miglior riferimento resta [12]. Qui ci limiteremo a descrivere brevemente un semplice esempio di connessione e le 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 un MSS 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) il 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 dal 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 [2] 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 MSL (*Maximum Segment Lifetime*).

La MSL è la stima del massimo periodo di tempo 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'header di IP (per maggiori dettagli vedi sez. A.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 di vita, si stima che un pacchetto IP non possa restare nella rete per più di MSL secondi.

Ogni implementazione del TCP deve scegliere un valore per la MSL (l'RFC 1122 raccomanda 2 minuti, Linux usa 30 secondi), questo comporta una durata dello stato **TIME\_WAIT** che a seconda delle implementazioni può variare fra 1 a 4 minuti. Lo stato **TIME\_WAIT** viene utilizzato dal protocollo per due motivi principali:

1. implementare 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 2MSL, in modo da 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, e 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. 14.3.2 e sez. 14.3.3 pure delle strutture degli indirizzi del socket.

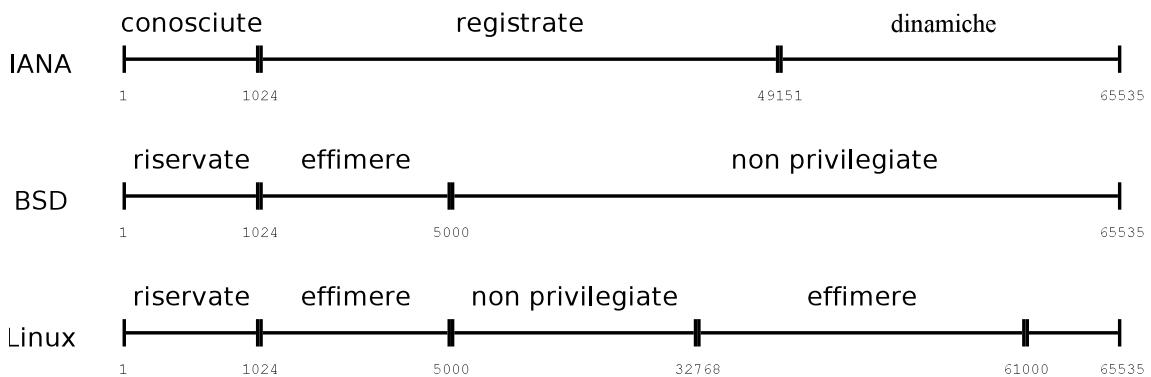
Quando un client contatta un server deve poter identificare con quale dei vari possibili server attivi intende parlare. Sia TCP che UDP definiscono un gruppo di *porte conosciute* (le cosiddette *well-known port*) che identificano una serie di servizi noti (ad esempio la porta 22 identifica il servizio SSH) effettuati da appositi server che rispondono alle connessioni verso tali porte.

D'altra parte un client non ha necessità di usare 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 <ftp://ftp.isi.edu/in-notes/iana/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 conosciute*. 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 implementato solo su TCP. Ad esempio X Window usa le porte TCP e UDP dal 6000 al 6063 anche se il protocollo è implementato solo tramite 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. Nel caso di Linux poi la scelta fra i due intervalli possibili viene fatta dinamicamente a seconda della memoria a disposizione del kernel per gestire le relative tabelle.



*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 conosciute). 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.

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 si suole chiamare *socket pair*<sup>4</sup> la combinazione dei quattro numeri che definiscono i due capi della connessione e cioè l'indirizzo IP locale e la porta TCP locale,

<sup>4</sup>da non confondere con la coppia di socket della omonima funzione `socketpair` che fanno riferimento ad una coppia di socket sulla stessa macchina, non ai capi di una connessione TCP.

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*.

### 15.1.7 Le porte ed il modello client/server

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 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<sup>5</sup> 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. 14.2.1.

### 15.2.1 La funzione bind

La funzione **bind** assegna un indirizzo locale ad un socket.<sup>6</sup> È usata cioè per specificare la prima parte della 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 restituisce 0 in caso di successo e -1 per un errore; in caso di errore la variabile **errno** viene impostata secondo i seguenti codici di errore:

<b>EBADF</b>	il file descriptor non è valido.
<b>EINVAL</b>	il socket ha già un indirizzo assegnato.
<b>ENOTSOCK</b>	il file descriptor non è associato ad un socket.
<b>EACCES</b>	si è cercato di usare una porta riservata senza sufficienti privilegi.
<b>EADDRNOTAVAIL</b>	Il tipo di indirizzo specificato non è disponibile.
<b>EADDRINUSE</b>	qualche altro socket sta già usando l'indirizzo.

ed anche **EFAULT** e per i socket di tipo **AF\_UNIX**, **ENOTDIR**, **ENOENT**, **ENOMEM**, **ELOOP**, **ENOSR** e **EROFS**.

<sup>5</sup>ad esempio con il comando **fuser**, o con **lsof**.

<sup>6</sup>nel 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.

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. 14.3.

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 server<sup>7</sup> 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 restrinerà 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). 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. 14.3.2, 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);
```

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'*endianess* 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`.

<sup>7</sup>un'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.

### 15.2.2 La funzione connect

La funzione **connect** è usata da un client TCP per stabilire la connessione con un server TCP,<sup>8</sup> il prototipo della funzione 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 restituisce zero in caso di successo e -1 per un errore, nel qual caso **errno** assumerà i valori:

**ECONNREFUSED** non c'è nessuno in ascolto sull'indirizzo remoto.

**ETIMEDOUT** si è avuto timeout durante il tentativo di connessione.

**ENETUNREACH** la rete non è raggiungibile.

**EINPROGRESS** il socket è non bloccante (vedi sez. 11.1.1) e la connessione non può essere conclusa immediatamente.

**EALREADY** il socket è non bloccante (vedi sez. 11.1.1) e un tentativo precedente di connessione non si è ancora concluso.

**EAGAIN** non ci sono più porte locali libere.

**EAFNOSUPPORT** l'indirizzo non ha una famiglia di indirizzi corretta nel relativo campo.

**EACCES**, **EPERM** si è tentato di eseguire una connessione ad un indirizzo broadcast senza che il socket fosse stato abilitato per il broadcast.

altri errori possibili sono: **EFAULT**, **EBADF**, **ENOTSOCK**, **EISCONN** e **EADDRINUSE**.

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. 14.3.

La struttura dell'indirizzo deve essere inizializzata con l'indirizzo IP e il numero di porta del server a cui ci si vuole connettere, come mostrato nell'esempio sez. 15.3.2, usando le funzioni illustrate in sez. 14.4.

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 sia con una opportuna **sysctl** che attraverso il filesystem **/proc** scrivendo il valore voluto in **/proc/sys/net/ipv4/tcp\_syn\_retries**. 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

<sup>8</sup>di nuovo 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 (nel caso del TCP il *three way handshake*) della connessione.

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. B.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 del 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;<sup>9</sup> 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:

```
#include <sys/socket.h>
int listen(int sockfd, int backlog)
    Pone un socket in attesa di una connessione.
```

La funzione restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

- `EBADF` l'argomento `sockfd` non è un file descriptor valido.
- `ENOTSOCK` l'argomento `sockfd` non è un socket.
- `EOPNOTSUPP` il socket è di un tipo che non supporta questa operazione.

La funzione pone il socket specificato da `sockfd` in modalità passiva e predisponde 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`.

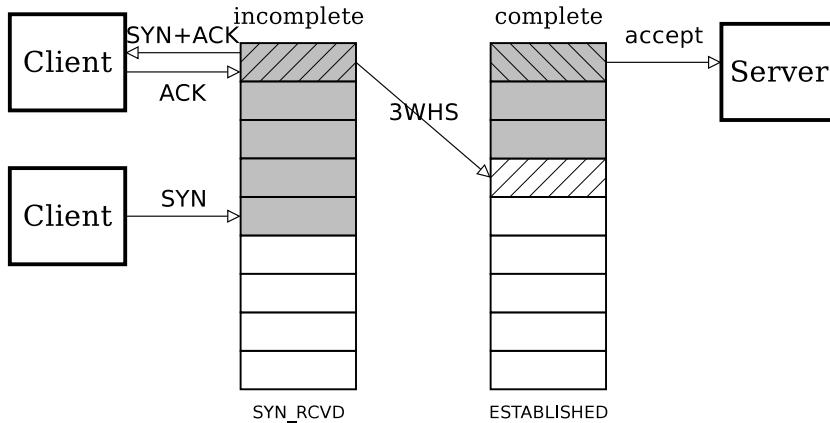
L'argomento `backlog` indica il numero massimo di connessioni pendenti accettate; se esso viene ecceduto il client al momento della richiesta della connessione riceverà un errore di tipo `ECONNREFUSED`, o se il protocollo, come accade nel caso del TCP, supporta la ritrasmissione, la richiesta sarà ignorata 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 un riferimento 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 un ingresso 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**).

<sup>9</sup> questa 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 SYN da un client il server crea una nuova voce nella coda delle connessioni incomplete, e poi risponde con il SYN+ACK. La voce resterà nella coda delle connessioni incomplete fino al ricevimento dell'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) la prima voce nella coda delle connessioni complete è passata al programma, o, se la coda è vuota, il processo viene posto in attesa e risvegliato all'arrivo della prima connessione completa.



**Figura 15.5:** Schema di funzionamento delle code delle connessioni complete ed incomplete.

Storicamente il valore del parametro `backlog` era corrispondente al massimo valore della somma del numero di voci possibili per ciascuna delle due code. Stevens in [2] 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 Linux 2.0, che mostrano le differenze fra diverse implementazioni.

In Linux il significato di questo valore è cambiato a partire dal kernel 2.2 per prevenire l'attacco chiamato *syn flood*. Questo si basa sull'emissione da parte dell'attaccante di un grande numero di pacchetti SYN indirizzati verso una porta, forgiati con indirizzo IP fasullo<sup>10</sup> così che i SYN+ACK vanno perduti e la coda delle connessioni incomplete viene saturata, impedendo di fatto ulteriori connessioni.

Per ovviare a questo il significato del `backlog` è stato cambiato a indicare la lunghezza della coda delle connessioni complete. La lunghezza della coda delle connessioni incomplete può essere ancora controllata usando la funzione `sysctl` con il parametro `NET_TCP_MAX_SYN_BACKLOG` o scrivendola direttamente in `/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 `/proc/sys/net/ipv4/tcp_synccookies`) questo valore viene ignorato e non esiste più un valore massimo. In ogni caso in Linux il valore di `backlog` viene troncato ad un massimo di `SOMAXCONN` se è superiore a detta costante (che di default vale 128).

La scelta storica per il valore di questo parametro 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. 2.3.4).

Stevens tratta accuratamente questo argomento in [2], con esempi presi da casi reali su web server, ed in particolare evidenzia come non sia più vero che il compito principale della coda sia

<sup>10</sup>con la tecnica che viene detta *ip spoofing*.

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 complete), 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 SYN arriva con tutte le code piene, il pacchetto deve essere ignorato. Questo perché la condizione in cui le code sono piene è ovviamente transitoria, per cui se il client ritrasmette il SYN è probabile che passato un po' di tempo possa trovare nella coda lo spazio per una nuova connessione. Se invece si rispondesse con un RST, per indicare l'impossibilità di effettuare la connessione, la chiamata a `connect` nel client ritornerebbe con una condizione di errore, costringendo a inserire nell'applicazione la gestione dei tentativi di riconnessione, che invece può essere 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 una volta che sia stato completato il *three way handshake*,<sup>11</sup> 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 restituisce un numero di socket descriptor positivo in caso di successo e -1 in caso di errore, nel qual caso `errno` viene impostata ai seguenti valori:

`EBADF` l'argomento `sockfd` non è un file descriptor valido.

`ENOTSOCK` l'argomento `sockfd` non è un socket.

`EOPNOTSUPP` il socket è di un tipo che non supporta questa operazione.

`EAGAIN` o `EWOULDBLOCK` il socket è stato impostato come non bloccante (vedi sez. 11.1.1), e non ci sono connessioni in attesa di essere accettate.

`EPERM` Le regole del firewall non consentono la connessione.

`ENOBUFS`, `ENOMEM` questo spesso significa che l'allocazione della memoria è limitata dai limiti sui buffer dei socket, non dalla memoria di sistema.

`EINTR` La funzione è stata interrotta da un segnale.

Inoltre possono essere restituiti gli errori di rete relativi al nuovo socket, diversi a seconda del protocollo, come: `EMFILE`, `EINVAL`, `ENOSR`, `ENOBUFS`, `EFAULT`, `EPERM`, `ECONNABORTED`, `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`. Nella struttura `addr` e nella variabile `addrlen` vengono restituiti indirizzo e relativa lunghezza del client che si è connesso.

I due argomenti `addr` e `addrlen` (si noti che quest'ultimo è passato per indirizzo per avere indietro il valore) sono usati per ottenere l'indirizzo del client da cui proviene la connessione. Prima della chiamata `addrlen` deve essere inizializzato alle dimensioni della struttura il cui indirizzo è passato come argomento in `addr`; al ritorno della funzione `addrlen` conterrà il numero di byte scritti dentro `addr`. Se questa informazione non interessa basterà inizializzare a `NULL` detti puntatori.

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

<sup>11</sup>la funzione è comunque generica ed è utilizzabile su socket di tipo `SOCK_STREAM`, `SOCK_SEQPACKET` e `SOCK_RDM`.

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 processo<sup>12</sup> 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,<sup>13</sup> 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`.

È da chiarire che Linux presenta un comportamento diverso nella gestione degli errori rispetto ad altre implementazioni 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`,<sup>14</sup> che devono essere rispecificati ogni volta. Tutto questo deve essere tenuto in conto se si devono scrivere programmi portabili.

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 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 funzioni più elementari sono queste, che vengono usate per ottenere i dati relativi alla socket pair associata ad un certo socket.

La prima funzione 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 restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

- |                       |  |
|-----------------------|--|
| <code>EBADF</code>    | l'argomento <code>sockfd</code> non è un file descriptor valido.       |
| <code>ENOTSOCK</code> | l'argomento <code>sockfd</code> non è un socket.                       |
| <code>ENOBUFS</code>  | non ci sono risorse sufficienti nel sistema per eseguire l'operazione. |
| <code>EFAULT</code>   | l'indirizzo <code>name</code> non è valido.                            |

La funzione restituisce la struttura degli indirizzi del socket `sockfd` nella struttura indicata dal puntatore `name` la cui lunghezza è specificata tramite l'argomento `namelen`. Quest'ultimo 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.

<sup>12</sup>a meno che non si sia impostato il socket per essere non bloccante (vedi sez. 11.1.1), nel qual caso ritorna con l'errore `EAGAIN`. Torneremo su questa modalità di operazione in sez. 15.6.

<sup>13</sup>attualmente in Linux solo DECnet ha questo comportamento.

<sup>14</sup>ed in generale tutti quelli che si possono impostare con `fcntl`, vedi sez. 6.3.5.

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 numero IP e 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.

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 restituisce 0 in caso di successo e -1 in caso di errore. I codici di errore restituiti in `errno` sono i seguenti:

<code>EBADF</code>	l'argomento <code>sockfd</code> non è un file descriptor valido.
<code>ENOTSOCK</code>	l'argomento <code>sockfd</code> non è un socket.
<code>ENOTCONN</code>	il socket non è connesso.
<code>ENOBUFS</code>	non ci sono risorse sufficienti nel sistema per eseguire l'operazione.
<code>EFAULT</code>	l'argomento <code>name</code> punta al di fuori dello spazio di indirizzi del processo.

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`.<sup>15</sup>

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,<sup>16</sup> 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 parametro, 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 implementazioni dei socket BSD fanno questa assunzione.

### 15.2.6 La funzione `close`

La funzione standard Unix `close` (vedi sez. 6.2.2) 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

<sup>15</sup>questa ad esempio è la modalità con cui opera il *super-server* `inetd`, che può gestire tutta una serie di servizi diversi, eseguendo su ogni connessione ricevuta sulle porte tenute sotto controllo, il relativo server.

<sup>16</sup>ad esempio il solito `inetd` fa sempre in modo che i file descriptor 0, 1 e 2 corrispondano al socket connesso.

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. 6.3.1, 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 implementa 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. 6.2.4 e sez. 6.2.5, che nel caso dei socket è particolarmente rilevante. Passeremo poi ad illustrare gli esempi dell'implementazione, sia dal lato client, che dal lato server, che si è realizzato 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. 6.2.4 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.

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. 12.1.1).

Per questo motivo, seguendo l'esempio di R. W. Stevens in [2], 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 più possibile leggere niente) e pertanto si ritorna senza aver concluso la lettura di tutti i byte

---

```

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.

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 `hton` per convertire il formato dell'intero usato dal computer a quello usato nella rete, infine 23-27 si può utilizzare la funzione `inet_nton` per convertire l'indirizzo numerico passato dalla linea di comando.

---

```

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.

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.

---

```

1 #include <sys/types.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
7 int main(int argc, char *argv[])
8 {
9     int sock_fd;
10    int i, nread;
11    struct sockaddr_in serv_addr;
12    char buffer[MAXLINE];
13    ...
14    /* create socket */
15    if ((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
16        perror("Socket creation error");
17        return -1;
18    }
19    /* initialize address */
20    memset((void *) &serv_addr, 0, sizeof(serv_addr)); /* clear server address */
21    serv_addr.sin_family = AF_INET;                      /* address type is INET */
22    serv_addr.sin_port = htons(13);                      /* daytime port is 13 */
23    /* build address using inet_nton */
24    if ((inet_nton(AF_INET, argv[optind], &serv_addr.sin_addr)) <= 0) {
25        perror("Address creation error");
26        return -1;
27    }
28    /* establish connection */
29    if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
30        perror("Connection error");
31        return -1;
32    }
33    /* read daytime from server */
34    while ((nread = read(sock_fd, buffer, MAXLINE)) > 0) {
35        buffer[nread]=0;
36        if (fputs(buffer, stdout) == EOF) {                /* write daytime */
37            perror("fputs error");
38            return -1;
39        }
40    }
41    /* error on read */
42    if (nread < 0) {
43        perror("Read error");
44        return -1;
45    }
46    /* normal exit */
47    return 0;
48 }
```

---

**Figura 15.8:** Esempio di codice di un client elementare per il servizio *daytime*.

Se abilitiamo il servizio *daytime*<sup>17</sup> possiamo verificare il funzionamento del nostro client, avremo allora:

```
[piccardi@gont sources]$ ./daytime 127.0.0.1
Mon Apr 21 20:46:11 2003
```

<sup>17</sup>in genere questo viene fornito direttamente dal superdemone *inetd*, pertanto basta assicurarsi che esso sia abilitato nel relativo file di configurazione.

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. 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 (27) 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 parametro, 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 un 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<sup>18</sup> per tener conto di quanto illustrato in sez. 10.1.5. 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.

---

<sup>18</sup>come una chiamata a `daemon` prima dell'inizio del ciclo principale.

---

```

1 #include <sys/types.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 <time.h>
7 #define MAXLINE 80
8 #define BACKLOG 10
9 int main(int argc, char *argv[])
10 {
11 /*
12 * Variables definition
13 */
14     int list_fd, conn_fd;
15     int i;
16     struct sockaddr_in serv_addr;
17     char buffer[MAXLINE];
18     time_t timeval;
19     ...
20     /* create socket */
21     if ((list_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
22         perror("Socket creation error");
23         exit(-1);
24     }
25     /* initialize address */
26     memset((void *)&serv_addr, 0, sizeof(serv_addr)); /* clear server address */
27     serv_addr.sin_family = AF_INET;                      /* address type is INET */
28     serv_addr.sin_port = htons(13);                      /* daytime port is 13 */
29     serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);       /* connect from anywhere */
30     /* bind socket */
31     if (bind(list_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
32         perror("bind error");
33         exit(-1);
34     }
35     /* listen on socket */
36     if (listen(list_fd, BACKLOG) < 0) {
37         perror("listen error");
38         exit(-1);
39     }
40     /* write daytime to client */
41     while (1) {
42         if ((conn_fd = accept(list_fd, (struct sockaddr *)NULL, NULL)) < 0) {
43             perror("accept error");
44             exit(-1);
45         }
46         timeval = time(NULL);
47         snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));
48         if ((write(conn_fd, buffer, strlen(buffer))) < 0) {
49             perror("write error");
50             exit(-1);
51         }
52         close(conn_fd);
53     }
54     /* normal exit */
55     exit(0);
56 }
```

---

*Figura 15.9:* Esempio di codice di un semplice server per il servizio daytime.

### 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 accennato anche in sez. 3.1 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'innesto 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 **atohs**.

---

```

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_add, 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)) < 0 ) {
23             perror("accept\u201derror");
24             exit(-1);
25         }
26         /* fork to handle connection */
27         if ( (pid = fork()) < 0 ){
28             perror("fork\u201derror");
29             exit(-1);
30         }
31         if (pid == 0) {                      /* child */
32             close(list_fd);
33             timeval = time(NULL);
34             snprintf(buffer, sizeof(buffer), "%.24s\r\n", ctime(&timeval));
35             if ( (write(conn_fd, buffer, strlen(buffer))) < 0 ) {
36                 perror("write\u201derror");
37                 exit(-1);
38             }
39             if (logging) {
40                 inet_ntop(AF_INET, &client.sin_addr, buffer, sizeof(buffer));
41                 printf("Request\u201dfrom\u201dhost\u201ds,\u201dport\u201d%d\u201d\n", buffer,
42                         ntohs(client.sin_port));
43             }
44             close(conn_fd);
45             exit(0);
46         } else {                            /* parent */
47             close(conn_fd);
48         }
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.

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.

## 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 implementazione 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 un'implementazione 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 [2], 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 il primo stabilisce che una volta stabilita la connessione ogni dato in ingresso deve essere rimandato in uscita fintanto che il chiamante non ha chiude 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*, disponibile nel file `TCP_echo_first.c`, 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 della 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.

---

```

1 int main(int argc, char *argv[])
2 {
3 /*
4 * Variables definition
5 */
6     int sock_fd, i;
7     struct sockaddr_in serv_addr;
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_addr, 0, sizeof(serv_addr)); /* clear server address */
16    serv_addr.sin_family = AF_INET; /* address type is INET */
17    serv_addr.sin_port = htons(7); /* echo port is 7 */
18    /* build address using inet_pton */
19    if ((inet_pton(AF_INET, argv[optind], &serv_addr.sin_addr)) <= 0) {
20        perror("Address creation error");
21        return 1;
22    }
23    /* establish connection */
24    if (connect(sock_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 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*.

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 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 riletti indietro (7) con una `read`<sup>19</sup> 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. 7.2.5) e la conseguente uscita dal ciclo; al che la subroutine ritorna ed il nostro programma client termina.

<sup>19</sup>si 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*.

Si può effettuare una verifica del funzionamento del client abilitando il servizio *echo* nella configurazione di `initd` sulla propria macchina ed usandolo direttamente verso di esso in 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. 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,<sup>20</sup> e poi si ripete (27-30) l'operazione usando `setuid` per cambiare anche l'utente.<sup>21</sup> 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 riprende di nuovo lo schema già visto usato dal server per

---

<sup>20</sup>si 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.

<sup>21</sup>si 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. 3.3) 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_addr;
6     ...
7     /* create socket */
8     if ((list_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
9         perror("Socket creation error");
10        exit(1);
11    }
12    /* initialize address and bind socket */
13    memset((void *)&serv_addr, 0, sizeof(serv_addr)); /* clear server address */
14    serv_addr.sin_family = AF_INET; /* address type is INET */
15    serv_addr.sin_port = htons(7); /* echo port is 7 */
16    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY); /* connect from anywhere */
17    if (bind(list_fd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
18        perror("bind error");
19        exit(1);
20    }
21    /* give away privileges and go daemon */
22    if (setgid(65534) != 0) { /* first give away group privileges */
23        perror("cannot give away group privileges");
24        exit(1);
25    }
26    if (setuid(65534) != 0) { /* and only after user ... */
27        perror("cannot give away user privileges");
28        exit(1);
29    }
30    if (demonize) { /* go daemon */
31        openlog(argv[0], 0, LOG_DAEMON); /* open logging */
32        if (daemon(0, 0) != 0) {
33            perror("cannot start as daemon");
34            exit(1);
35        }
36    }
37    /* main body */
38    if (listen(list_fd, BACKLOG) < 0) { /* listen on socket */
39        PrintErr("listen error");
40        exit(1);
41    }
42    while (1) { /* handle echo to client */
43        len = sizeof(cli_addr);
44        if ((conn_fd = accept(list_fd, NULL, NULL)) < 0) {
45            PrintErr("accept error");
46            exit(1);
47        }
48        if ((pid = fork()) < 0) { /* fork to handle connection */
49            PrintErr("fork error");
50            exit(1);
51        }
52        if (pid == 0) { /* child */
53            close(list_fd); /* close listening socket */
54            ServEcho(conn_fd); /* handle echo */
55            exit(0);
56        } else { /* parent */
57            close(conn_fd); /* close connected socket */
58        }
59    }
60    exit(0); /* normal exit, never reached */
61 }

```

---

**Figura 15.13:** Codice del corpo principale della prima versione del server per il servizio *echo*.

il servizio *daytime*, con l'unica differenza della chiamata alla funzione `PrintErr`, riportata in fig. 15.14, al posto di `perror` per la stampa degli errori.

Si inizia con il porre (37-41) in ascolto il socket, e poi si esegue indefinitamente il ciclo principale (42-59). All'interno di questo si ricevono (43-47) le connessioni, creando (48-51) un processo figlio per ciascuna di esse. Quest'ultimo (52-56), chiuso (53) il *listening socket*, esegue (54) la funzione di gestione del servizio `ServEcho`, ed al ritorno di questa esce (55).

Il padre invece si limita (57) 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. 10.1.5. Come accennato questo è stato fatto utilizzando come *wrapper* la funzione `PrintErr`, il cui codice è riportato in fig. 15.14.

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. 10.1.5) per stampare il messaggio 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.

---

```

1 void PrintErr(char * error) {
2     if (demonize) {                                /* daemon mode */
3         syslog(LOG_ERR, "%s: %m", error); /* log string and error message */
4     } else {
5         perror(error);
6     }
7     return;
8 }
```

---

**Figura 15.14:** Codice della funzione `PrintErr` per la generalizzazione della stampa degli errori sullo standard input o attraverso il `syslog`.

La gestione del servizio *echo* viene effettuata interamente nella funzione `ServEcho`, il cui codice è mostrato in fig. 15.15, 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 successo) si provvede (8-10) a stampare il relativo messaggio con `PrintErr`. Quando il client chiude la connessione il ricevimento del 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 normale

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 normali, 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.

---

```

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 15.15:** Codice della prima versione della funzione `ServEcho` per la gestione del servizio `echo`.

Il primo passo è compilare e lanciare il server (da root, per poter usare la porta 7 che è riservata), alla partenza 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 client<sup>22</sup> 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
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:

<sup>22</sup>si 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.

```
[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. 3.5).

Se a questo punto si inizia a scrivere qualcosa sul client non sarà trasmesso niente fin tanto che non si prema il tasto di a capo (si ricordi quanto detto in sez. 7.2.5 a proposito dell'I/O su terminale), solo allora `fgets` ritornerà ed il client scriverà quanto immesso 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'immediatamente 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 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. 3.2.4). 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. 9.3.6), 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. 3.2.5), cosa che faremo utilizzando **SIGCHLD** secondo quanto illustrato in sez. 9.3.6. 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. 9.10), per installare il gestore che riceve i segnali dei processi figli terminati già visto in fig. 9.4. 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. 9.3.1, 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. 9.3.1 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. 9.10. Definiremo allora la nuova funzione **SignalRestart**<sup>23</sup> come mostrato in fig. 15.16, ed installeremo il gestore usando quest'ultima.

Come si può notare questa funzione è identica alla precedente **Signal**, illustrata in fig. 9.10, 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

---

<sup>23</sup>anche 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.

---

```

1 inline SigFunc * SignalRestart(int signo, SigFunc *func)
2 {
3     struct sigaction new_handl, old_handl;
4     new_handl.sa_handler = func;           /* set signal handler */
5     new_handl.sa_flags = SA_RESTART;       /* restart system call */
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    /* change action for signo signal */
11    if (sigaction(signo, &new_handl, &old_handl)){
12        return SIG_ERR;
13    }
14    return (old_handl.sa_handler);
15 }
```

---

**Figura 15.16:** La funzione `SignalRestart`, che installa un gestore di segnali in semantica BSD per il riavvio automatico delle system call interrotte.

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,<sup>24</sup> 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. 15.17, 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.

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. 15.17 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

---

<sup>24</sup>Stevens in [2] 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.

---

```

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_addr);
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            sprintf(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 15.17:** La sezione nel codice della seconda versione del server per il servizio *echo* modificata per tener conto dell'interruzione delle system call.

fig. 15.17; 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 critica, 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 *slow system call*<sup>25</sup> 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. 15.15, 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`<sup>26</sup> 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 miglioria 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. 15.18.

---

```

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 (demonize) { /* daemon mode */
20            syslog(LOG_DEBUG, debug);
21        } else {
22            printf("%s", debug);
23        }
24    }
25 }
26 return;
27 }
```

---

**Figura 15.18:** Codice della seconda versione della funzione `ServEcho` per la gestione del servizio *echo*.

Rispetto alla precedente versione di fig. 15.15 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.

<sup>25</sup>si ricordi la distinzione fatta in sez. 9.3.1.

<sup>26</sup>la 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`.

Inoltre qualora sia stata attivata la funzionalità di debug (avvalorando `debugging` tramite l'apposita opzione `-d`) si provvederà a stampare (tenendo conto della modalità di invocazione del server, se interattiva o in forma di demone) il numero di byte e la stringa letta dal client (16-24).

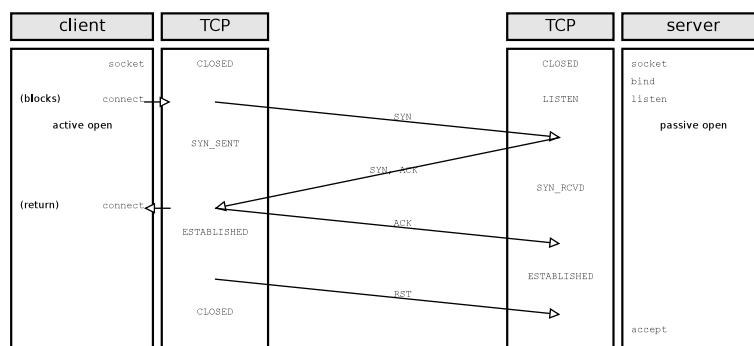
## 15.5 I vari scenari critici

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 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. 15.19, 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 15.19:** 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. 15.19, 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`, che come nel caso precedente, necessitano semplicemente la ripetizione della chiamata senza che si debba 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 di RST alla chiusura dal socket (come suggerito da Stevens in [2]), non si ha 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; 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<sup>27</sup> 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 da 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 *advertising 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 scriviamo 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'echo nel client che lo stamperà a video.

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)
```

---

<sup>27</sup>in realtà si è ridotta la lunghezza dell'output rispetto al reale tagliando alcuni dati non necessari alla comprensione del flusso.

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`<sup>28</sup>), 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 analoghi a delle pipe. Allora, da quanto illustrato in sez. 12.1.1, 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. 15.20.

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

<sup>28</sup>il comando `strace` è un comando di debug molto utile che prende come parametro un altro comando e ne stampa a video tutte le invocazioni di una system call, coi relativi parametri 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.

---

```

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 15.20:** La sezione nel codice della seconda versione della funzione `ClientEcho` usata dal client per il servizio *echo* modificata per tener conto degli eventuali errori.

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 diventa 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 crollo della rete, cosa che potremo simulare

facilmente staccando il cavo di rete. Un'altra condizione è quella di un blocco della macchina completo della su cui gira il server che deve essere riavviata, cosa che potremo simulare sia premendo il bottone di reset,<sup>29</sup> che, in maniera più gentile, riavviando la macchina dopo aver interrotto la connessione di rete.

Cominciamo ad analizzare il primo caso, il crollo della rete. Ripetiamo la nostra sessione di lavoro precedente, lanciamo il client, scriviamo una prima riga, poi stacchiamo 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
gont.34685 > anarres.echo: P 12:45(33) ack 12 win 5840
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 ottendo nessuna risposta. Quello che succede è che nel frattempo il kernel

---

<sup>29</sup>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.

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 dall'implementazione 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. 16.3.3). Questo parametro infatti specifica il numero di volte che deve essere rientrata 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*<sup>30</sup> 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 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
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
```

<sup>30</sup>la *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.

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 stacchiamo 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 ricevuti 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. 11.1, 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. 11.1 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. 11.1 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 errori pendenti su un socket usando l'opzione `SO_ERROR`.
- quando si sta utilizzando un *listening socket* ed ci sono delle connessioni complete. In questo caso la funzione `accept` non si bloccherà.<sup>31</sup>

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:

- 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

---

<sup>31</sup>in 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.

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 *fuori banda* (o *out-of-band*), una caratteristica specifica dei socket TCP su cui torneremo in sez. 18.1.1.

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.<sup>32</sup>

### 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 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).

Riprendiamo allora il codice del client, modificandolo per l'uso di `select`. Quello che dobbiamo modificare è la funzione `ClientEcho` di fig. 15.20, dato che tutto il resto, che riguarda le modalità in cui viene stabilita la connessione con il server, resta assolutamente identico. La

---

<sup>32</sup>questo 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 sà 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 file descriptor, pur avendo disponibili dei dati, non viene dato per pronto in lettura.

---

```

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 15.21:** La sezione nel codice della terza versione della funzione `ClientEcho` usata dal client per il servizio `echo` modificata per l'uso di `select`.

nostra nuova versione di `ClientEcho`, la terza della serie, è riportata in fig. 15.21, 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 relativi, 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 sono passati tutti puntatori nulli, non interessando 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 una 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 (27-30) si è avuto un errore e pertanto si esce immediatamente segnalandolo, se è nullo (31-34) significa che il server ha chiuso la connessione, e di nuovo si esce con stampando prima un messaggio di avviso, 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 ridiventava 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 ridiventava 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.<sup>33</sup>

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 *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 rileggere quanto l'altro capo può continuare a scrivere. Per poter permettere allora di segnalare che si è concluso con la scrittura, continuando al contempo a leggere quanto può provenire dall'altro capo del socket si può allora 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 restituisce zero in caso di successo e -1 per un errore, nel qual caso `errno` assumerà i valori:

`ENOTSOCK` il file descriptor non corrisponde a un socket.

`ENOTCONN` il socket non è connesso.

ed inoltre `EBADF`.

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:

<sup>33</sup>abbiamo incontrato questa situazione nei vari scenari critici di sez. 15.5.

<b>SHUT_RD</b>	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.
<b>SHUT_WR</b>	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 <i>half-close</i> . Tutti i dati presenti nel buffer di scrittura prima della chiamata saranno inviati, seguiti dalla sequenza di chiusura illustrata in sez. 15.1.3.
<b>SHUT_RDWR</b>	chiude sia il lato in lettura che quello in scrittura del socket. È equivalente alla chiamata in sequenza con <b>SHUT_RD</b> e <b>SHUT_WR</b> .

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. 15.21, 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. 15.22 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. 15.21; 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 subordinato 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 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.

---

```

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 scrittura: %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 scrittura su terminale");
47                return;
48            }
49        }
50    }
51 }
```

---

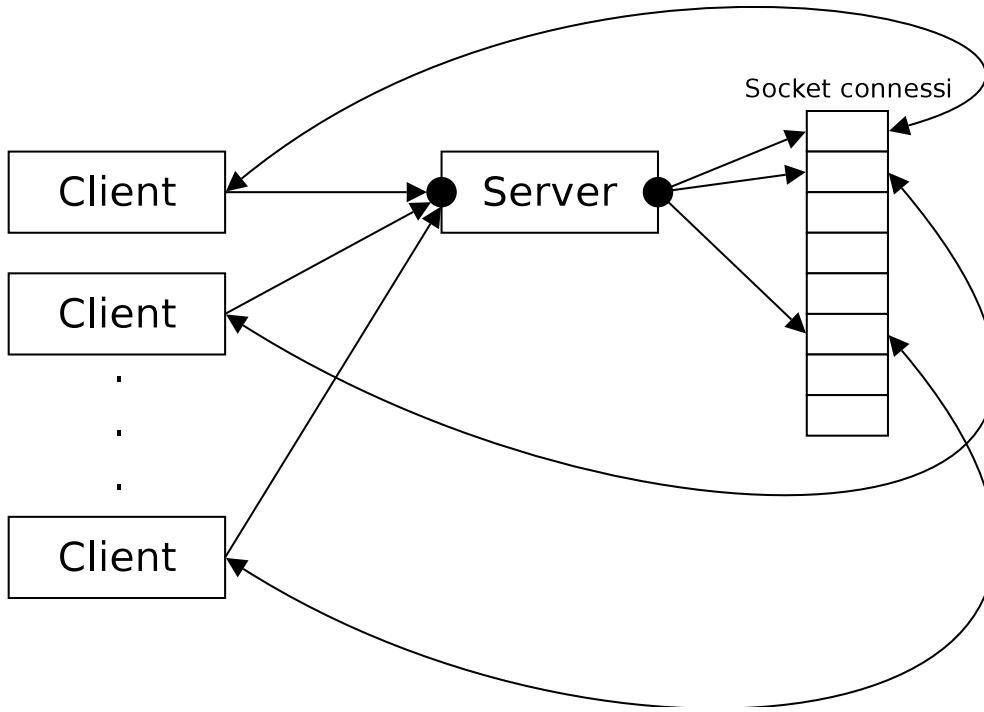
**Figura 15.22:** La sezione nel codice della versione finale della funzione ClientEcho, che usa shutdown per una conclusione corretta della connessione.

#### 15.6.4 Un server basato sull'I/O multiplexing

Seguendo di nuovo le orme di Stevens in [2] vediamo ora come con l'utilizzo dell'I/O multiplexing diventi possibile riscrivere completamente il nostro server *echo* con una architettura completa-

mente diversa, in modo da evitare di dover creare un nuovo processo tutte le volte che si ha una connessione.<sup>34</sup>

La struttura del nuovo server è illustrata in fig. 15.23, 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 15.23:** Schema del nuovo server echo basato sull'I/O multiplexing.

La sezione principale del codice del nuovo server è illustrata in fig. 15.24. 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_echod.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 implementare lo schema mostrato in fig. 15.23, 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 (6-56) 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,<sup>35</sup> che verrà anche (4) inserito nella tabella.

La prima sezione (7-10) 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 nella tabella *fd\_open*. Dato che la chiamata di *select* modifica il valore del *file descriptor set*, è

<sup>34</sup>ne faremo comunque una implementazione diversa rispetto a quella presentata da Stevens in [2].

<sup>35</sup>in quanto esso è l'unico file aperto, oltre i tre standard, e pertanto avrà il valore più alto.

---

```

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   /* main loop, wait for connection and data inside a select */
6   while (1) {
7       FD_ZERO(&fset);                /* clear fd_set */
8       for (i = list_fd; i <= max_fd; i++) { /* initialize fd_set */
9           if (fd_open[i] != 0) FD_SET(i, &fset);
10      }
11      while ((n = select(max_fd + 1, &fset, NULL, NULL, NULL)) < 0)
12          && (errno == EINTR));           /* wait for data or connection */
13      if (n < 0) {                   /* on real error exit */
14          PrintErr("select_error");
15          exit(1);
16      }
17      if (FD_ISSET(list_fd, &fset)) { /* if new connection */
18          n--;                      /* decrement active */
19          len = sizeof(c_addr);      /* and call accept */
20          if ((fd = accept(list_fd, (struct sockaddr *)&c_addr, &len)) < 0) {
21              PrintErr("accept_error");
22              exit(1);
23          }
24          fd_open[fd] = 1;            /* set new connection socket */
25          if (max_fd < fd) max_fd = fd; /* if needed set new maximum */
26      }
27      /* loop on open connections */
28      i = list_fd;                /* first socket to look */
29      while (n != 0) {            /* loop until active */
30          i++;                    /* start after listening socket */
31          if (fd_open[i] == 0) continue; /* closed, go next */
32          if (FD_ISSET(i, &fset)) { /* if active process it */
33              n--;
34              nread = read(i, buffer, MAXLINE); /* read operations */
35              if (nread < 0) {
36                  PrintErr("Errore in lettura");
37                  exit(1);
38              }
39              if (nread == 0) { /* if closed connection */
40                  close(i);        /* close file */
41                  fd_open[i] = 0;    /* mark as closed in table */
42                  if (max_fd == i) { /* if was the maximum */
43                      while (fd_open[--i] == 0); /* loop down */
44                      max_fd = i;        /* set new maximum */
45                      break;             /* and go back to select */
46                  }
47                  continue;         /* continue loop on open */
48              }
49              nwrite = FullWrite(i, buffer, nread); /* write data */
50              if (nwrite) {
51                  PrintErr("Errore in scrittura");
52                  exit(1);
53              }
54          }
55      }
56  }
57  ...

```

---

**Figura 15.24:** La sezione principale del codice della nuova versione di server echo basati sull'uso della funzione select.

necessario ripetere (7) ogni volta il suo azzeramento, per poi procedere con il ciclo (8-10) in cui si impostano i socket trovati attivi.

Per far questo si usa la caratteristica dei file descriptor, descritta in sez. 6.2.1, 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 (8) 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 (9) nella nostra tabella se il file descriptor è in uso o meno,<sup>36</sup> 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 eseguiremo (11-12) 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 (13-16) ad uscire stampando un messaggio di errore.

Se invece la funzione ritorna normalmente avremo in `n` il numero di socket da controllare. Nello specifico si danno due possibili casi diversi per cui `select` può essere ritornata: 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 (17-26): si controlla (17) che il socket in ascolto sia fra quelli attivi, nel qual caso anzitutto (18) se ne decrementa il numero in `n`; poi, inizializzata (19) la lunghezza della struttura degli indirizzi, si esegue `accept` per ottenere il nuovo socket connesso controllando che non ci siano errori (20-23). 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 (24) il nuovo file descriptor alla tabella di quelli connessi, ed inoltre, se è il caso, aggiornare (25) il valore massimo in `max_fd`.

Una volta controllato l'arrivo di nuove connessioni si passa a verificare se vi sono dati sui socket connessi, per questo si ripete un ciclo (29-55) fintanto che il numero di socket attivi `n` resta diverso da zero; in questo modo se l'unico socket con attività era quello connesso, avendo opportunamente decrementato il contatore, il ciclo verrà saltato, e si ritornerà immediatamente (ripetuta l'inizializzazione del file descriptor set con i nuovi valori nella tabella) alla chiamata di `accept`. 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 comunque si inizializza (28) 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 (30) 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 (31) se questo è nella tabella dei socket connessi, chiedendo la ripetizione del ciclo in caso contrario. Altrimenti si passa a verificare (32) se il file descriptor corrisponde ad uno di quelli attivi, e nel caso si esegue (33) una lettura, uscendo con un messaggio in caso di errore (34-38).

Se (39) il numero di byte letti `nread` è nullo si è in presenza del caso di un *end-of-file*, indice che una connessione che si è chiusa, che deve essere trattato (39-48) opportunamente. Il primo passo è chiudere (40) anche il proprio capo del socket e rimuovere (41) il file descriptor dalla tabella di quelli aperti, inoltre occorre verificare (42) se il file descriptor chiuso è quello con il valore più alto, nel qual caso occorre trovare (42-46) il nuovo massimo, altrimenti (47) 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

---

<sup>36</sup>si tenga presente che benché il kernel assegni sempre il primo valore libero, dato che nelle operazioni i socket saranno aperti e chiusi in corrispondenza della creazione e conclusione delle connessioni, si potranno sempre avere dei buchi nella nostra tabella.

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 (43) che trova il primo valore per cui la tabella presenta un file descriptor aperto, e lo imposta (44) come nuovo massimo, per poi tornare (44) 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 (49) **FullWrite** per riscriverli indietro sul socket stesso, avendo cura di uscire con un messaggio in caso di errore (50–53). Si noti che nel ciclo si esegue una sola lettura, contrariamente a quanto fatto con la precedente versione (si riveda il codice di fig. 15.18) 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 risolvibili 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. 11.1 come la funzione **poll** possa costituire una alternativa a **select**, con alcuni vantaggi.<sup>37</sup>

Ancora una volta in sez. 11.1.3 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 *out-of-band* 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.

---

<sup>37</sup>non soffrendo delle limitazioni dovute all'uso dei *file descriptor set*.

- 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 l'implementazione la classifica come normale.

Come esempio dell'uso di **poll** proviamo allora a reimplementare il server *echo* secondo lo schema di fig. 15.23 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. 15.25 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 (5) l'osservazione 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, output ed 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 10-55) che ha una struttura sostanzialmente identica a quello usato per il precedente esempio basato su **select**. La prima istruzione (11-12) 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 (13-16) alla terminazione immediata del processo provvedendo a stampare una descrizione dello stesso.

Una volta ottenuta dell'attività su un file descriptor si hanno di nuovo due possibilità. La prima possibilità 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 (18) a decrementare la variabile **n** (che assume il significato di numero di file descriptor attivi rimasti da controllare) per poi (19-23) effettuare la chiamata ad **accept**, terminando il processo in caso di errore. Se la chiamata ad **accept** ha successo si procede attivando (24) la struttura relativa al nuovo file descriptor da essa ottenuto, modificando (24) 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 (27) l'indice **i** del vettore delle strutture **pollfd** al valore del socket in

---

```

1  /* initialize all needed variables */
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++) {
5      poll_set[i].fd = -1;
6      poll_set[i].events = POLLRDNORM;
7  }
8  poll_set[max_fd].fd = list_fd;
9  /* main loop, wait for connection and data inside a select */
10 while (1) {
11     while ((n = poll(poll_set, max_fd + 1, -1)) < 0)
12         && (errno == EINTR));                  /* wait for data or connection */
13     if (n < 0) {                           /* on real error exit */
14         PrintErr("poll_error");
15         exit(1);
16     }
17     if (poll_set[list_fd].revents & POLLRDNORM) { /* if new connection */
18         n--;                                /* decrement active */
19         len = sizeof(c_addr);                /* and call accept */
20         if ((fd = accept(list_fd, (struct sockaddr *)&c_addr, &len)) < 0) {
21             PrintErr("accept_error");
22             exit(1);
23         }
24         poll_set[fd].fd = fd;                 /* set new connection socket */
25         if (max_fd < fd) max_fd = fd;        /* if needed set new maximum */
26     }
27     i = list_fd;                          /* first socket to look */
28     while (n != 0) {                     /* loop until active */
29         i++;                            /* start after listening socket */
30         if (poll_set[i].fd == -1) continue; /* closed, go next */
31         if (poll_set[i].revents & (POLLRDNORM|POLLERR)) {
32             n--;                        /* decrease active */
33             nread = read(i, buffer, MAXLINE); /* read operations */
34             if (nread < 0) {
35                 PrintErr("Errore in lettura");
36                 exit(1);
37             }
38             if (nread == 0) {            /* if closed connection */
39                 close(i);              /* close file */
40                 poll_set[i].fd = -1;    /* mark as closed in table */
41                 if (max_fd == i) {      /* if was the maximum */
42                     while (poll_set[--i].fd == -1); /* loop down */
43                     max_fd = i;          /* set new maximum */
44                     break;              /* and go back to select */
45                 }
46                 continue;            /* continue loop on open */
47             }
48             nwrite = FullWrite(i, buffer, nread); /* write data */
49             if (nwrite) {
50                 PrintErr("Errore in scrittura");
51                 exit(1);
52             }
53         }
54     }
55 }
56 exit(0);      /* normal exit, never reached */

```

---

**Figura 15.25:** La sezione principale del codice della nuova versione di server echo basati sull'uso della funzione poll.

ascolto, dato che gli ulteriori socket aperti avranno comunque un valore superiore. Il ciclo (28-54) 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 (30) l'indice e controllare (31) 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 (31) 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 (32) a decrementare il numero `n` di file descriptor attivi da controllare e ad eseguire (33) la lettura, ed in caso di errore (34-37) al solito lo si notificherà uscendo immediatamente. Qualora invece si ottenga una condizione di end-of-file (38-47) si provvederà a chiudere (39) anche il nostro capo del socket e a marcarlo (40) nella struttura ad esso associata come inutilizzato. Infine dovrà essere ricalcolato (41-45) un eventuale nuovo valore di `max_fd`. L'ultimo passo è (46) chiudere 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 (48) ad effettuarne la riscrittura all'indietro, con il solito controllo ed eventuale uscita e notifica in caso di errore (49-52).

Come si può notare la logica del programma è identica a quella vista in fig. 15.24 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. 14.4 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*

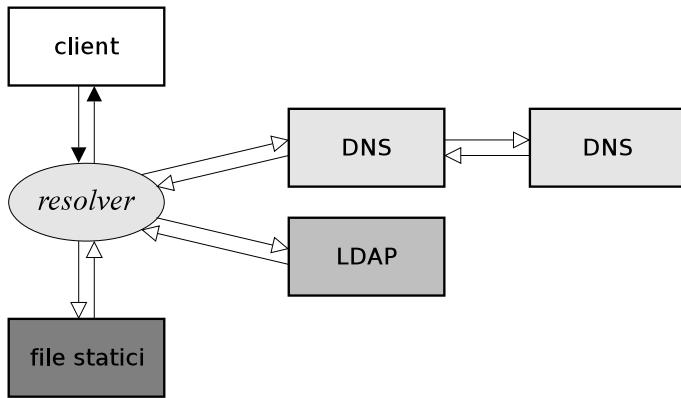
La risoluzione dei nomi è associata tradizionalmente al servizio del *Domain Name Service* che permette di identificare le macchine su internet invece che per numero IP attraverso il relativo *nome a dominio*.<sup>1</sup> In realtà per DNS si intendono spesso i server che forniscono su internet questo servizio, mentre nel nostro caso affronteremo la problematica dal lato client, di un qualunque programma che necessita di compiere questa operazione.

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. 8.2.3 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 routine fornite con le librerie del C, detto appunto *resolver*.

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.<sup>2</sup>

<sup>1</sup>non staremo ad entrare nei dettagli della definizione di cosa è un nome a dominio, dandolo per noto, una introduzione alla problematica si trova in [13] (cap. 9) mentre per una trattazione approfondita di tutte le problematiche relative al DNS si può fare riferimento a [14].

<sup>2</sup>la sigla LDAP fa riferimento ad un protocollo, il *Lightweight Directory Access Protocol*, che prevede un



**Figura 16.1:** Schema di funzionamento delle routine del *resolver*.

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 affianca il file `/etc/host.conf` il cui scopo principale è 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 associazioni 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,<sup>3</sup> 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.<sup>4</sup>

Per risolvere questa serie di problemi la risoluzione dei nomi a dominio eseguirà 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*<sup>5</sup> cui abbiamo accennato anche in sez. 8.2.3 per quanto riguarda la gestione dei dati associati a utenti e gruppi. 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

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.

<sup>3</sup>il *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.

<sup>4</sup>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).

<sup>5</sup>il sistema è stato introdotto la prima volta nelle librerie standard di Solaris, le *glibc* hanno ripreso lo stesso schema, si tenga presente che questo sistema non esiste per altre librerie standard come le *libc5* o le *uclib*.

fra nomi e valori numerici, sia l'ordine in cui effettuare le ricerche 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
<code>shadow</code>	corrispondenze fra username e proprietà dell'utente ( <code>uid</code> , ecc.).
<code>group</code>	corrispondenze fra nome del gruppo e proprietà dello stesso.
<code>aliases</code>	alias per la posta elettronica
<code>ethers</code>	corrispondenze fra numero IP e MAC address della scheda di rete.
<code>hosts</code>	corrispondenze fra nome a dominio e numero IP.
<code>netgroup</code>	corrispondenze gruppo di rete e macchine che lo compongono.
<code>networks</code>	corrispondenze fra nome di una rete e suo indirizzo IP.
<code>protocols</code>	corrispondenze fra nome di un protocollo e relativo numero identificativo.
<code>rpc</code>	corrispondenze fra nome di un servizio RPC e relativo numero identificativo.
<code>services</code>	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<sup>6</sup> di configurazione per ciascuna di queste classi, che viene inizia 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.

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 fornire 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,<sup>7</sup> e sono queste quelle che tratteremo nelle sezioni successive.

### 16.1.2 Le funzioni di interrogazione del *resolver*

Prima di trattare le funzioni usate normalmente nella risoluzione dei nomi a dominio conviene trattare in maniera più dettagliata il meccanismo principale da esse utilizzato e cioè quello del servizio DNS. Come accennato questo, benché 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. 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 per realizzare le funzioni generiche di libreria usate anche 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

<sup>6</sup>seguendo una convezione comune per i file di configurazione le righe vuote vengono ignorate e tutto quello che segue un carattere "#" viene considerato un commento.

<sup>7</sup>è cura della implementazione fattane nelle `glibc` tenere conto della presenza del *Name Service Switch*.

.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.<sup>8</sup> 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 sottodominio di terzo livello ad un altro server ancora.

In realtà un server DNS è in grado di fare 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.<sup>9</sup> 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, rispetto a quelle relative alla semplice risoluzione dei nomi, un insieme di funzioni specifiche dedicate all'interrogazione di un server DNS; 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 restituisce 0 in caso di successo e -1 in caso di errore.
```

La funzione legge il contenuto dei file di configurazione (i già citati **resolv.conf** e **host.conf**) 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 direttamente in quanto viene automaticamente chiamata la prima volta che si esegue una delle altre.

Le impostazioni e lo stato del *resolver* vengono mantenuti in una serie di variabili raggruppate nei campi di una apposita struttura **\_res** usata da tutte queste funzioni. Essa viene definita in **resolv.h** ed è utilizzata internamente alle funzioni essendo definita come variabile globale; questo consente anche di accedervi direttamente all'interno di un qualunque programma, una volta che la sia opportunamente dichiarata come:

```
extern struct state _res;
```

Tutti i campi della struttura sono ad uso interno, e vengono usualmente inizializzati da **res\_init** 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**, una maschera binaria che contiene una serie di bit di opzione che permettono di controllare il comportamento del *resolver*.

Per utilizzare questa funzionalità per modificare le impostazioni direttamente da programma occorrerà impostare un opportuno valore per questo campo ed invocare esplicitamente **res\_init**, 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:

<sup>8</sup>per 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.

<sup>9</sup>ritroveremo classi di indirizzi e tipi di record più avanti in tab. 16.3 e tab. 16.4.

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 "."). usato con <code>RES_USEVC</code> per mantenere aperte le connessioni TCP fra interrogazioni diverse.
RES_STAYOPEN	se attivo <code>res_search</code> esegue le ricerche di nomi di macchine nel dominio corrente o nei domini ad esso sovrastanti.
RES_DNSRCH	blocca i controlli di sicurezza di tipo 1.
RES_INSECURE1	blocca i controlli di sicurezza di tipo 2.
RES_INSECURE2	blocca l'uso della variabile di ambiente <code>HOSTALIASES</code> .
RES_NOALIASES	restituisce indirizzi IPv6 con <code>gethostbyname</code> .
RES_USE_INET6	ruota la lista dei server DNS dopo ogni interrogazione.
RES_ROTATE	non controlla i nomi per verificarne la correttezza sintattica.
RES_NOCHECKNAME	non elimina i record di tipo TSIG.
RES_KEEPTSIG	
RES_BLAST	
RES_DEFAULT	è l'insieme di <code>RES_RECURSE</code> , <code>RES_DEFNAMES</code> e <code>RES_DNSRCH</code> .

**Tabella 16.2:** Costanti utilizzabili come valori per `_res.options`.

```
#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 soprassiede 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 soprassiede il valore del campo `retrans`,<sup>10</sup> 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`, 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 suo 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)
Esegue una interrogazione al DNS.
```

La funzione restituisce un valore positivo pari alla lunghezza dei dati scritti nel buffer `answer` in caso di successo e -1 in caso di errore.

<sup>10</sup>preimpostato al valore della omonima costante `RES_TIMEOUT` di `resolv.h`.

La funzione esegue 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 addizionali previste dalle due opzioni `RES_DEFNAMES` e `RES_DNSRCH`, è `res_search`, 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)
Esegue una interrogazione al DNS.
```

La funzione restituisce un valore positivo pari alla lunghezza dei dati scritti nel buffer `answer` in caso di successo e -1 in caso di errore.

In sostanza la funzione ripete una serie di chiamate a `res_query` aggiungendo al nome contenuto nella stringa `dname` il dominio di default da cercare, fermendosi 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 [14].

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.<sup>11</sup>

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<sup>12</sup> 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,<sup>13</sup> 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 sottosinsieme, 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 entreremo nei dettagli

<sup>11</sup>esisteva in realtà anche una classe `C_CSNET` per la omonima rete, ma è stata dichiarata obsoleta.

<sup>12</sup>ripreso dai file di dichiarazione `arpa/nameser.h` e `arpa/nameser_compat.h`.

<sup>13</sup>BIND, 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 instrarare 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`.

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 `dodds.truelite.it` e l'indirizzo IP `62.48.34.25`.
- 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`, o `sources.truelite.it`, che fanno sempre riferimento a `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
<code>HOST_NOT_FOUND</code>	l'indirizzo richiesto non è valido e la macchina indicata è sconosciuta.
<code>NO_ADDRESS</code>	il nome a dominio richiesto è valido, ma non ha un indirizzo associato ad esso (alternativamente può essere indicato come <code>NO_DATA</code> ).
<code>NO_RECOVERY</code>	si è avuto un errore non recuperabile nell'interrogazione di un server DNS.
<code>TRY AGAIN</code>	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 due nuove funzioni per stampare l'errore a video, analoghe a quelle di sez. 8.5.2 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'analogia 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'analogia `strerror`, restituisce una stringa con un messaggio di errore già formattato, corrispondente al codice passato come argomento (che si presume sia dato da `h_errno`).

### 16.1.3 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 di richiesta generica ed esamineremo invece le funzioni a questo dedicate. La prima funzione 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 al nome a dominio name.
```

La funzione restituisce in caso di successo il puntatore ad una struttura di tipo `hostent` contenente i dati associati al nome a dominio, o un puntatore nullo in caso di 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. A.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` normalmente 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<sup>14</sup> 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 di tipo af associato al nome a dominio name.

```

La funzione restituisce in caso di successo il puntatore ad una struttura di tipo `hostent` contenente i dati associati al nome a dominio, o un puntatore nullo in caso di errore.

In questo caso la funzione prende un secondo argomento `af` che indica (i soli valori consentiti sono `AF_INET` o `AF_INET6`, per questo è necessario l'uso di `sys/socket.h`) la famiglia di indirizzi

<sup>14</sup>questa è una estensione fornita dalle *glibc*, disponibile anche in altri sistemi unix-like.

che dovrà essere utilizzata nei risultati restituiti dalla funzione. Per tutto il resto la funzione è identica a `gethostbyname`, ed identici sono i suoi risultati.

---

```

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    if ((argc - optind) != 1) {
13        printf("Wrong number of arguments %d\n", argc - optind);
14        usage();
15    }
16    data = gethostbyname(argv[1]);
17    if (data == NULL) {
18        perror("Errore di risoluzione");
19        exit(1);
20    }
21    printf("Canonical name %s\n", data->h_name);
22    alias = data->h_aliases;
23    while (*alias != NULL) {
24        printf("Alias %s\n", *alias);
25        alias++;
26    }
27    if (data->h_addrtype == AF_INET) {
28        printf("Address are IPv4\n");
29    } else if (data->h_addrtype == AF_INET6) {
30        printf("Address are IPv6\n");
31    } else {
32        printf("Tipo di indirizzo non valido\n");
33        exit(1);
34    }
35    alias = data->h_addr_list;
36    while (*alias != NULL) {
37        addr = inet_ntop(data->h_addrtype, *alias, buffer, sizeof(buffer));
38        printf("Indirizzo %s\n", addr);
39        alias++;
40    }
41    exit(0);
42 }
```

---

**Figura 16.3:** Esempio di codice per la risoluzione di un indirizzo.

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 (12-15) esce con un errore. Dopo di che (16) si limita a chiamare `gethostbyname`, ricevendo il risultato nel puntatore `data`. Questo (17-20) viene controllato per rilevare eventuali errori, nel qual caso il programma esce dopo aver stampato un messaggio con `perror`.

Se invece la risoluzione è andata a buon fine si inizia (21) con lo stampare il nome canonico, dopo di che (22–26) si stampano eventuali altri nomi. Per questo prima (22) si prende il puntatore alla cima della lista che contiene i nomi e poi (23–26) si esegue un ciclo che sarà ripetuto fin tanto che nella lista si troveranno dei puntatori validi<sup>15</sup> per le stringhe dei nomi; prima (24) si stamperà la stringa e poi (25) 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 (27–34) 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 (35–40) si stamperanno i valori degli indirizzi, di nuovo (35) si inizializzerà 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 (37) ad una conversione con la funzione `inet_ntop` (vedi sez. 14.4) passandole gli opportuni argomenti, questa restituirà la stringa da stampare (38) 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.<sup>16</sup>

Le funzioni illustrate finora hanno un difetto: utilizzando una area di memoria interna per allocare i contenuti della struttura `hostent` non possono essere rientranti. Questo 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 sovrascritti; per questo motivo, se si vuole salvare il risultato di una chiamata, occorrerà eseguire quella che si chiama una *deep copy*.<sup>17</sup>

Per ovviare a questi problemi nelle *glibc* sono definite anche delle versioni rientranti delle 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_errno)
int gethostbyname2_r(const char *name, int af, struct hostent *ret, char *buf,
    size_t buflen, struct hostent **result, int *h_errno)
Versioni rientranti delle funzioni gethostbyname e gethostbyname2.
```

Le funzioni restituiscono 0 in caso di successo ed un valore negativo in caso di 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

<sup>15</sup>si ricordi che la lista viene terminata da un puntatore nullo.

<sup>16</sup>volendo essere pignoli si dovrebbe controllarne 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.

<sup>17</sup>si 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 sottostrutture con altri puntatori) e copiare anche i dati da questi referenziati.

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 codice di errore negativo 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<sup>18</sup> sono previste delle funzioni apposite; la prima è `sethostent`, il cui prototipo è:

```
#include <netdb.h>
void sethostent(int stayopen)
    Richiede l’uso di connessioni per le interrogazioni ad un server DNS.
```

La funzione non restituisce nulla.

La funzione 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 pari a 1 (o diverso da zero), che indica una condizione vera in C, attiva la funzionalità. Come si attiva l’uso delle connessioni TCP lo si può disattivare con la funzione `endhostent`; il suo prototipo è:

```
#include <netdb.h>
void endhostent(void)
    Disattiva l’uso di connessioni per le interrogazioni ad un server DNS.
```

La funzione non restituisce nulla.

e come si può vedere 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 restituisce l’indirizzo ad una struttura `hostent` in caso di successo ed `NULL` in caso di 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`<sup>19</sup> per un indirizzo IPv4 ed una struttura `in6_addr` per un indirizzo IPv6, mentre in `len` se ne dovrà

<sup>18</sup>si potrebbero impostare direttamente le opzioni di `__res.options`, ma queste funzioni permettono di semplificare la procedura.

<sup>19</sup>si ricordi che, come illustrato in fig. 14.2, questo in realtà corrisponde ad un numero intero, da esprimere comunque in *network order*, non altrettanto avviene però per `in6_addr`, pertanto è sempre opportuno inizializzare questi indirizzi con `inet_pton` (vedi sez. 14.4.4).

specificare la dimensione (rispettivamente 4 o 16), infine l'argomento `type` indica 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 funziona comunque inizializza anche il primo campo della lista `h_addr_list` col valore dell'indirizzo passato come argomento.

Per risolvere il problema dell'uso da parte delle due funzioni `gethostbyname` e `gethostbyaddr` 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`), vennero introdotte due nuove funzioni di risoluzione,<sup>20</sup> `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)
struct hostent *getipnodebyaddr(const void *addr, size_t len, int af, int
    *error_num)
Richiedono rispettivamente la risoluzione e la risoluzione inversa di un indirizzo IP.
```

Entrambe le funzioni restituiscono l'indirizzo ad una struttura `hostent` in caso di successo ed `NULL` in caso di 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.

Costante	Significato
<code>AI_V4MAPPED</code>	usato con <code>AF_INET6</code> per richiedere una ricerca su un indirizzo IPv4 invece che IPv6; gli eventuali risultati saranno rimappati su indirizzi IPv6.
<code>AI_ALL</code>	usato con <code>AI_V4MAPPED</code> ; richiede sia indirizzi IPv4 che IPv6, e gli indirizzi IPv4 saranno rimappati in IPv6.
<code>AI_ADDRCONFIG</code>	richiede che una richiesta IPv4 o IPv6 venga eseguita solo se almeno una interfaccia del sistema è associata ad un indirizzo di tale tipo.
<code>AI_DEFAULT</code>	il valore di default, è equivalente alla combinazione di <code>AI_ADDRCONFIG</code> e di <code>AI_V4MAPPED</code> .

**Tabella 16.6:** Valori possibili per i bit dell'argomento `flags` della funzione `getipnodebyname`.

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

<sup>20</sup>le funzioni sono presenti nelle *glibc* versione 2.1.96, ma essendo considerate deprecate (vedi sez. 16.1.4) sono state rimosse nelle versioni successive.

`gethostbyaddr`. L'uso di una allocazione dinamica però comporta anche la necessità di deallocare 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:

```
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
void freehostent(struct hostent *ip)
    Disalloca una struttura hostent.

La funzione non ritorna nulla.
```

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` 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	getservbyaddr
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*.

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 NSS 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 noti (cioè `http`, `smtp`, ecc.) da associare alle rispettive porte, le due funzioni da utilizzare per questo sono `getservbyname` e `getservbyaddr`, 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.
```

Ritornano il puntatore ad una struttura `servent` con i risultati in caso di successo, o `NULL` in caso di errore.

Entrambe le funzioni prendono come ultimo argomento una stringa `proto` che indica il protocollo per il quale si intende effettuare la ricerca,<sup>21</sup> che nel caso si IP può avere come valori

<sup>21</sup>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.

possibili solo `udp` o `tcp`;<sup>22</sup> 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`<sup>23</sup> ed estraggono 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 hanno una struttura del tutto analoga alle precedenti, e 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. 8.10 per le informazioni relative ai dati degli utenti e dei gruppi. Nel caso specifico dei servizi avremo allora le tre funzioni `setservent`, `getservent` e `endservent` i cui prototipi sono:

```
#include <netdb.h>
void setservent(int stayopen)
    Apre il file /etc/services e si posiziona al suo inizio.
struct servent *getservent(void)
    Legge la voce successiva nel file /etc/services.
void endservent(void)
    Chiude il file /etc/services.
```

Le due funzioni `setservent` e `endservent` non restituiscono nulla, `getservent` restituisce il puntatore ad una struttura `servent` in caso di successo e `NULL` in caso di 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.

<sup>22</sup>in teoria si potrebbe avere un qualunque protocollo fra quelli citati in `/etc/protocols`, posto che lo stesso supporta il concetto di *porta*, in pratica questi due sono gli unici presenti.

<sup>23</sup>il *Name Service Switch* astrae il concetto a qualunque supporto su cui si possano mantenere i suddetti dati.

L'argomento `stayopen`, se diverso da zero, fa sì che il file resti aperto anche fra diverse chiamate a `getservbyname` e `getservbyaddr`.<sup>24</sup> 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.

Informazione	Funzioni		
indirizzo	<code>sethostent</code>	<code>gethostent</code>	<code>endhostent</code>
servizio	<code>cd tesetservernt</code>	<code>getservernt</code>	<code>endservernt</code>
rete	<code>setnetent</code>	<code>getnetent</code>	<code>endnetent</code>
protocollo	<code>setprotoent</code>	<code>getprotoent</code>	<code>endprotoent</code>

*Tabella 16.8:* Funzioni lettura sequenziale dei dati del *Name Service Switch*.

#### 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,<sup>25</sup> comunque esse non forniscono una interfaccia sufficientemente generica.

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 deprecated le varie funzioni `gethostbyaddr`, `gethostbyname`, `getipnodebyname` e `getipnodebyaddr` ed è stata introdotta una interfaccia completamente nuova.

La prima funzione di questa interfaccia è `getaddrinfo`,<sup>26</sup> 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; il suo prototipo è:

```
#include <netdb.h>
#include <sys/socket.h>
#include <netdb.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 restituisce 0 in caso di successo o un codice di errore diverso da zero in caso di fallimento.

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.

<sup>24</sup>di default dopo una chiamata a queste funzioni il file viene chiuso, cosicché una successiva chiamata a `getservernt` riparte dall'inizio.

<sup>25</sup>rimane 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.

<sup>26</sup>la funzione è definita, insieme a `getnameinfo` che vedremo più avanti, nell'RFC 2553.

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.

---

```
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.

La struttura `addrinfo`, la cui definizione<sup>27</sup> 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. 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; 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 `PF_INET` e `PF_INET6`, mentre se non si vuole specificare nessuna famiglia di indirizzi si può usare il valore `PF_UNSPEC`. Allo stesso modo per `ai_socktype` si possono usare i valori illustrati in sez. 14.2.3 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

<sup>27</sup>la definizione è ripresa direttamente dal file `netdb.h` in 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.

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 l'ultimo campo è `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.

Costante	Significato
<code>AI_PASSIVE</code>	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 <code>loopback</code> . Se invece non è impostato gli indirizzi verranno restituiti in formato adatto ad una chiamata a <code>connect</code> o <code>sendto</code> .
<code>AI_CANONNAME</code>	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> .
<code>AI_NUMERICHOST</code>	se impostato il nome della macchina specificato con <code>node</code> deve essere espresso in forma numerica, altrimenti sarà restituito un errore <code>EAI_NODATA</code> (vedi tab. 16.10), in questo modo si evita ogni chiamata alle funzioni di risoluzione.
<code>AI_V4MAPPED</code>	stesso significato dell'analogia di tab. 16.6.
<code>AI_ALL</code>	stesso significato dell'analogia di tab. 16.6.
<code>AI_ADDRCONFIG</code>	stesso significato dell'analogia di tab. 16.6.

**Tabella 16.9:** Costanti associate ai bit del campo `ai_flags` della struttura `addrinfo`.

Come ultimo argomento di `getaddrinfo` 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 in `res` il puntatore iniziale ad una *linked list* di strutture di tipo `addrinfo` contenenti tutte le informazioni ottenute.

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 chiama 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`.

Come per i codici di errore di `gethostbyname` anche in questo caso è fornita una apposita funzione, analoga di `strerror`, che consente di utilizzarla direttamente 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 restituisce il puntatore alla stringa contenente il messaggio di errore.

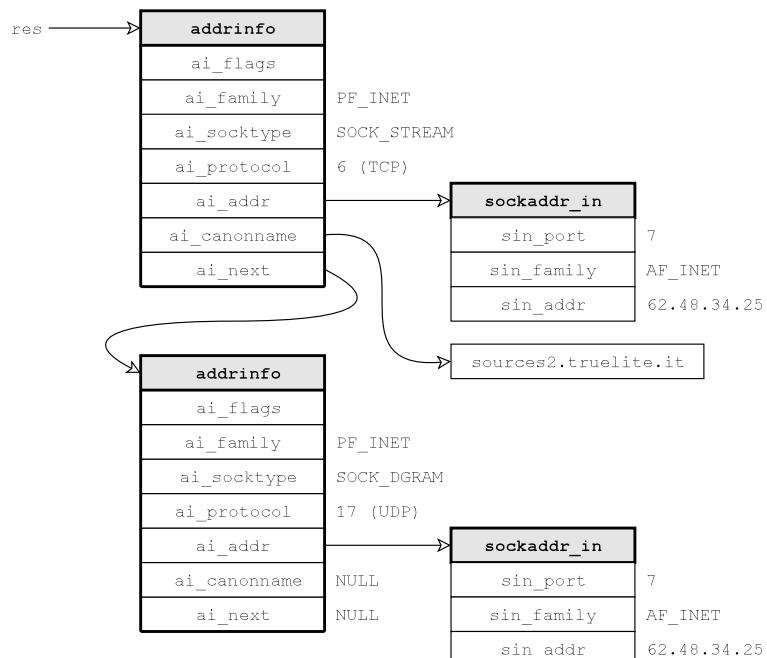
La funzione restituisce un puntatore alla stringa contenente il messaggio corrispondente dal

Costante	Significato
EAI_FAMILY	la famiglia di indirizzi richiesta non è supportata.
EAI_SOCKTYPE	il tipo di socket richiesto non è supportato.
EAI_BADFLAGS	il campo <code>ai_flags</code> contiene dei valori non validi.
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_ADDRFAMILY	la rete richiesta non ha nessun indirizzo di rete per la famiglia di indirizzi specificata.
EAI_NODATA	la macchina specificata esiste, ma non ha nessun indirizzo di rete definito.
EAI_MEMORY	è stato impossibile allocare la memoria necessaria alle operazioni.
EAI_FAIL	il DNS ha restituito un errore di risoluzione permanente.
EAI AGAIN	il DNS ha restituito un errore di risoluzione temporaneo, si può ritentare in seguito.
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`.

codice di errore `errcode` ottenuto come valore di ritorno di `getaddrinfo`. La stringa è allocata staticamente, ma essendo costante, ed accessibile in sola lettura, questo non comporta nessun problema di rientranza della funzione.

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.

---

```

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;                                /* init list pointer */
13 printf("Canonical name %s\n", ptr->ai_canonname); /* print cname */
14 while (ptr != NULL) {                      /* loop on list */
15     if (ptr->ai_family == PF_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 == PF_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.

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.<sup>28</sup>

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` si sia dovuto passare il puntatore al campo contenente l'indirizzo IP nella struttura puntata dal campo `ai_addr`.<sup>29</sup>

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 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`.

<sup>28</sup>questa eventualità non dovrebbe mai verificarsi, almeno fintanto che la funzione `getaddrinfo` lavora correttamente.

<sup>29</sup>il 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 costrutto `&addr6->sin6_addr` è corretto in quanto l'operatore `->` ha su questo caso precedenza su `&`).

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 **getservname** è **getnameinfo**, ed il suo prototipo è:

```
#include <sys/socket.h>
#include <netdb.h>
int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t
    hostlen, char *serv, size_t servlen, int flags)
Risolve il contenuto di una struttura degli indirizzi in maniera indipendente dal protocollo.
```

La funzione restituisce 0 in caso di successo e un codice di errore diverso da zero altrimenti.

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**. Si può, quando non si è interessati ad uno dei due, passare il valore NULL come argomento, così che la corrispondente informazione non verrà 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.

Costante	Significato
<b>NI_NOFQDN</b>	richiede che venga restituita solo il nome della macchina all'interno del dominio al posto del nome completo (FQDN).
<b>NI_NUMERICHOST</b>	richiede che venga restituita la forma numerica dell'indirizzo (questo succede sempre se il nome non può essere ottenuto).
<b>NI_NAMEREQD</b>	richiede la restituzione di un errore se il nome non può essere risolto.
<b>NI_NUMERICSERV</b>	richiede che il servizio venga restituito in forma numerica (attraverso il numero di porta).
<b>NI_DGRAM</b>	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.

**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**<sup>30</sup> 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**, provvederemo una interfaccia semplificata per i due casi visti finora, quello in cui si specifica nel client un indirizzo

<sup>30</sup>In Linux le due costanti sono definite in **netdb.h** ed hanno rispettivamente il valore 1024 e 12.

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.

---

```

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:\u2014resolution\u2014failed:");
13        fprintf(stderr, "\u2014%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:\u2014cannot\u2014create\u2014socket");
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:\u2014cannot\u2014connect");
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`.

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. 14.2.3). 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<sup>31</sup> si sono stampati i messaggi d'errore direttamente nella funzione.

Una volta definite le variabili necessarie (3-5) la funzione prima (6) azzerà 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`.

---

```

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        return 1;
12    }
13    /* do read/write operations */
14    ClientEcho(stdin, sock);
15    /* normal exit */
16    return 0;
17 }
```

---

**Figura 16.9:** Il nuovo codice per la connessione del client echo.

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

<sup>31</sup>non si può avere nessuna certezza che detti valori siano negativi, è questo è invece necessario per evitare ogni possibile ambiguità nei confronti del valore di ritorno in caso di successo.

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 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`.

La seconda funzione di ausilio 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.

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) {                                /* install signal handler */
10         Signal(SIGCHLD, HandSigCHLD);           /* non restarting handler */
11     } else {
12         SignalRestart(SIGCHLD, HandSigCHLD);    /* 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 siano trattati allo stesso modo dei file, ed acceduti tramite i file descriptor, la normale interfaccia usata per la gestione dei file non è sufficiente a poterne controllare tutte le caratteristiche, che variano tra l'altro a seconda del loro tipo (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, le cosiddette *socket options*.

### 16.2.1 Le funzioni `setsockopt` e `getsockopt`

Le varie caratteristiche dei socket possono essere gestite attraverso l'uso di due funzioni 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 restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i 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.

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`.<sup>32</sup>

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`,<sup>33</sup> è la dimensione in byte dei dati presenti all'indirizzo indicato da `optval`. Dato che

<sup>32</sup>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_*` (citeate anche da Stevens in [2]); 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`.

<sup>33</sup>questo 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 dalle `glibc`.

Livello	Significato
SOL_SOCKET	opzioni generiche dei socket.
SOL_IP	opzioni specifiche per i socket che usano IPv4.
SOL_TCP	opzioni per i socket che usano TCP.
SOL_IPV6	opzioni specifiche per i socket che usano IPv6.
SOL_ICMPV6	opzioni specifiche per i socket che usano ICMPv6.

**Tabella 16.12:** Possibili valori dell'argomento `level` delle funzioni `setsockopt` e `getsockopt`.

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, am 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 restituisce 0 in caso di successo e -1 in caso di errore, nel qual caso `errno` assumerà i valori:

<code>EBADF</code>	il file descriptor <code>sock</code> non è valido.
<code>EFAULT</code>	l'indirizzo <code>optval</code> o quello di <code>optlen</code> non è valido.
<code>ENOPROTOOPT</code>	l'opzione scelta non esiste per il livello indicato.
<code>ENOTSOCK</code>	il file descriptor <code>sock</code> 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 qualsiasi tipo di socket,<sup>34</sup> indipendentemente da quale protocollo venga poi utilizzato. Se si vuole operare su queste opzioni generiche il livello da utilizzare è `SOL_SOCKET`; si è riportato un elenco di queste opzioni in tab. 16.13.

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`)

<sup>34</sup>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`.

Opzione	get	set	flag	Tipo	Descrizione
SO_KEEPALIVE	•	•	•	int	controlla l'attività della connessione.
SO_OOBINLINE	•	•	•	int	lascia in linea i dati <i>out-of-band</i> .
SO_RCVLOWAT	•	•	•	int	basso livello sul buffer di ricezione.
SO SNDLOWAT	•	•		int	basso livello sul buffer di trasmissione.
SO_RCVTIMEO	•	•		timeval	timeout in ricezione.
SO SNDTIMEO	•	•		timeval	timeout in trasmissione.
SO_BSDCOMPAT	•	•	•	int	abilita la compatibilità con BSD.
SO_PASSCRED	•	•	•	int	abilita la ricezione di credenziali.
SO_PEERCREDS	•			ucred	restituisce le credenziali del processo remoto.
SO_BINDTODEVICE	•	•		char *	lega il socket ad un dispositivo.
SO_DEBUG	•	•	•	int	abilita il debugging sul socket.
SO_REUSEADDR	•	•	•	int	consente il riutilizzo di un indirizzo locale.
SO_TYPE	•			int	restituisce il tipo di socket.
SO_ACCEPTCONN	•			int	indica se il socket è in ascolto.
SO_DONTROUTE	•	•	•	int	non invia attraverso un gateway.
SO_BROADCAST	•	•	•	int	attiva o disattiva il <i>broadcast</i> .
SO_SNDBUF	•	•		int	imposta dimensione del buffer di trasmissione.
SO_RCVBUF	•	•		int	imposta dimensione del buffer di ricezione.
SO_LINGER	•	•		linger	indugia nella chiusura con dati da spedire.
SO_PRIORITY	•	•		int	imposta la priorità del socket.
SO_ERROR	•			int	riceve e cancella gli errori pendenti.

**Tabella 16.13:** Le opzioni disponibili al livello SOL\_SOCKET.

l'opzione è disponibile, mentre la colonna successiva indica, quando di 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.

Dato che le descrizioni di tab. 16.13 sono estremamente sommarie, vale la pena entrare in dettagli maggiori; questo ci consentirà anche di trattare i vari casi particolari, dato che nonostante queste opzioni siano indicate al livello generico, alcune di esse han senso solo per alcuni tipi di socket. L'elenco dettagliato del significato di ciascuna di esse è allora il seguente:

**SO\_KEEPALIVE** una connessione può restare attiva se non viene effettuato alcun traffico su di essa; in certi casi però può essere utile controllarne lo stato per accorgersi di eventuali problemi. Per questo, se si imposta questa opzione, è 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 socket che supportino le connessioni (non ha senso per socket UDP ad esempio), ed utilizza per `optval` un intero usato come valore logico.

L'opzione si applica principalmente ai socket TCP. Con le impostazioni di default (che sono riprese da BSD) Linux emette un messaggio di *keep-alive* 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; tutto ciò viene effettuato dal kernel e le applicazioni non riceveranno nessun dato.

In caso di problemi invece si possono avere i due casi già illustrati in sez. 15.5.3 per il caso di terminazione precoce del server: il primo è quello in cui la macchina remota è caduta ed è stata riavviata, per cui dopo il riavvio la connessione non viene più riconosciuta,<sup>35</sup> e si otterrà come risposta un RST.

<sup>35</sup>si ricordi che un normale riavvio non ha questo effetto, in quanto si passa per la chiusura del processo, che chiude anche il socket inviando un segmento FIN all'altro capo della connessione.

In tal caso il socket viene chiuso dopo aver impostato un errore **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 ad un massimo di 9 volte<sup>36</sup> (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**. Se invece si riceve in risposta ad uno di questi messaggi un pacchetto ICMP di destinazione irraggiungibile, verrà restituito l'errore corrispondente.

In generale questa opzione serve per individuare una caduta della connessione,<sup>37</sup> e viene usata sui server per evitare di mantenere impegnate le risorse dedicate a trattare delle connessioni in realtà terminate. Abilitandola le connessioni effettivamente terminate vengono chiuse ed una **select** potrà rilevare la conclusione delle stesse e ricevere il relativo errore. Si tenga però presente che non si ha la certezza assoluta che un errore di **ETIMEDOUT** corrisponda ad una reale conclusione della connessione, il problema potrebbe essere dovuto ad un problema di routing che perduri per un tempo maggiore di quello impiegato nei vari tentativi di ritrasmissione del *keep-alive*.

#### **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. 18.1.1. 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\_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, ma con Linux questo valore è sempre 1 e non può essere cambiato; **getsockopt** leggerà questo valore mentre **setsockopt** darà un errore di **ENOPROTOOPT**.

#### **SO SNDLOWAT**

questa opzione imposta il valore che indica il numero minimo di byte che devono essere presenti nel buffer di scrittura 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, come per la precedente **SO\_RCVLOWAT** con Linux questo valore è sempre 1 e non può essere cambiato; **getsockopt** leggerà questo valore mentre **setsockopt** darà un errore di **ENOPROTOOPT**.

#### **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. 8.9) 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.

<sup>36</sup>entrambi questi valori possono essere opportunamente modificati con gli opportuni parametri illustrati in sez. 16.3.3, si tenga presente che però questo vale a livello di kernel ed i valori saranno applicati a *tutti* i socket.

<sup>37</sup>il crash di un processo di nuovo comporta la chiusura di tutti i file che aveva aperti e la relativa emissione degli opportuni segmenti FIN nel caso dei socket.

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.<sup>38</sup>

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.

**SO\_SNDFTIMEO** 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`, `writev`, `send`, `sendfrom` e `sendmsg` qualora queste restino bloccate per un tempo maggiore di quello specificato.

**SO\_BSDCOMPAT** questa opzione abilita la compatibilità con il comportamento di BSD (in particolare ne riproduce i bug). Attualmente è una opzione usata solo per il protocollo UDP e ne è prevista la rimozione in futuro. 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 per i socket raw, che sono state rimosse con il passaggio al 2.2; è consigliato correggere i programmi piuttosto che usare questa funzione.

**SO\_PASSCRED** questa opzione abilita sui socket unix-domain la ricezione dei messaggi di controllo di tipo `SCM_CREDENTIALS`. Prende come `optval` un intero usato come valore logico.

**SO\_PEERCRED** questa opzione restituisce le credenziali del processo remoto connesso al socket; l'opzione è disponibile solo per socket unix-domain e può essere usata solo con `getsockopt`. Utilizza per `optval` una apposita struttura `ucred` (vedi sez. ??).

**SO\_BINDTODEVICE** questa opzione permette di legare il socket ad una particolare interfaccia 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`); se si utilizza una stringa nulla o un valore nullo per `optlen` si rimuove 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. ??).

**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

<sup>38</sup>in 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; dato che con Linux questo valore è sempre 1 questo caso non esiste.

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;<sup>39</sup> quando viene abilitata una serie di messaggi con le informazioni di debug vengono inviati direttamente al sistema del kernel log.<sup>40</sup>

#### **SO\_REUSEADDR**

questa opzione permette di eseguire la funzione **bind** su indirizzi locali che siano già in uso; 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<sup>41</sup> è già in uso da parte di un altro socket.

Come Stevens sottolinea in [2] si distinguono quattro casi per l'utilizzo di questa opzione; il primo 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; quando si riavvia il server questo viene bloccato sulla chiamata a **bind** dato che la porta è ancora utilizzata in una connessione esistente.<sup>42</sup> Inoltre se si usa il protocollo TCP questo può avvenire anche dopo che l'ultimo processo figlio è terminato, dato che la connessione può restare attiva anche dopo la chiusura del socket mantenendosi nello stato TIME\_WAIT.

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 estremamente remota,<sup>43</sup> che eventuali pacchetti rimasti intrappolati in una precedente connessione possano finire fra quelli di una nuova.

Il secondo caso in cui viene usata questa opzione è quando si ha una macchina cui sono assegnati diversi numeri 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 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 risulti libera. Infine si tenga presente che con il protocollo TCP non è mai possibile far partire server che eseguano **bind** sullo stesso indirizzo e la stessa porta, cioè ottenere quello che viene chiamato un *completely duplicate binding*.

---

<sup>39</sup>deve 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**.

<sup>40</sup>si 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**.

<sup>41</sup>più propriamente il controllo viene eseguito sulla porta.

<sup>42</sup>questa è una delle domande più frequenti sui newsgroup dedicati allo sviluppo, in quanto è piuttosto comune in questa situazione quando si sta sviluppando un server che si ferma e si riavvia in continuazione.

<sup>43</sup>perché 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.

Il terzo impiego è simile al precedente e prevede l'uso di `bind` all'interno dello stesso programma per associare indirizzi diversi a socket diversi. Vale in questo caso quanto detto in precedenza, l'unica differenza è che in questo caso le diverse chiamate a `bind` sono eseguite all'interno dello stesso programma.

<code>SO_TYPE</code>	questa opzione permette di leggere il tipo di socket su cui si opera; funziona solo con <code>getsockopt</code> , ed utilizza per <code>optval</code> un intero in cui verrà restituito il valore numerico che lo identifica (ad esempio <code>SOCK_STREAM</code> ).
<code>SO_ACCEPTCONN</code>	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 <code>listen</code> . L'opzione può essere usata soltanto con <code>getsockopt</code> e utilizza per <code>optval</code> un intero in cui viene restituito 1 se il socket è in ascolto e 0 altrimenti.
<code>SO_DONTROUTE</code>	questa opzione forza l'invio diretto dei pacchetti del socket, saltando ogni processo relativo all'uso della tabella di routing del kernel. Prende come <code>optval</code> un intero usato come valore logico.
<code>SO_BROADCAST</code>	
<code>SO_SNDBUF</code>	
<code>SO_RCVBUF</code>	
<code>SO_LINGER</code>	
<code>SO_PRIORITY</code>	
<code>SO_ERROR</code>	

## 16.3 Altre funzioni di controllo

Benché la maggior parte delle caratteristiche dei socket sia gestita attraverso le due funzioni `setsockopt` e `getsockopt`, alcune funzionalità possono essere impostate attraverso quelle che sono le funzioni classiche per il controllo delle proprietà dei file, cioè `fcntl` e `ioctl`.

### 16.3.1 L'uso di `fcntl` per i socket

Abbiamo già trattato l'uso di `fcntl` in sez. 6.3.5, dove però ne abbiamo descritto le funzionalità nell'ambito della sua applicazione a file descriptor associati a file normali; tratteremo qui invece il suo uso specifico quando la si impiega su file descriptor associati a dei socket.

### 16.3.2 L'uso di `ioctl` per i socket

Come per `fcntl` abbiamo trattato l'uso di `ioctl` in sez. 6.3.6, dove ne abbiamo descritto le funzionalità nell'ambito dell'applicazione su file normali; tratteremo qui il suo uso specifico quando la si impiega su file descriptor associati a dei socket.

### 16.3.3 L'uso di `sysctl` per le proprietà della rete

Come ultimo argomento di questa sezione tratteremo l'uso della funzione `sysctl` (che è stata introdotta nelle sue funzionalità generiche in sez. 8.2.1) per quanto riguarda le sue capacità di effettuare impostazioni relative a proprietà generali dei socket (di tutti quelli di un certo tipo o di tutti quelli che usano un certo protocollo) rispetto alle funzioni viste finora che consentono di controllare quelle di un singolo socket.



# 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. 12.1.5.

### 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. B.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.

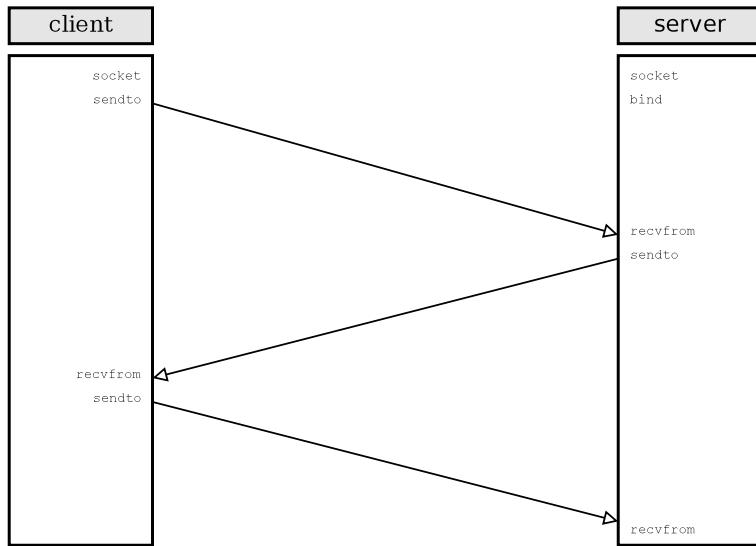
Questo diverso comportamento significa anche che i socket UDP, pur appartenendo alla famiglia `PF_INET`<sup>1</sup> devono essere aperti quando si usa la funzione `socket` (si riveda quanto illustrato a suo tempo in tab. 14.2) utilizzando per il tipo di socket il valore `SOCK_DGRAM`.

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.

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`.

---

<sup>1</sup>o `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.

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 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 ha neanche a disposizione un socket connesso su cui sia possibile usare direttamente `read` e `write` avendo già stabilito (grazie

alla chiamata ad `accept` che lo associa ad una connessione) quali sono sorgente e destinazione dei dati.

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 prototipo<sup>2</sup> 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:

<code>EAGAIN</code>	il socket è in modalità non bloccante, ma l'operazione richiede che la funzione si blocchi.
<code>ECONNRESET</code>	l'altro capo della comunicazione ha resettato la connessione.
<code>EDESTADDRREQ</code>	il socket non è di tipo connesso, e non si è specificato un indirizzo di destinazione.
<code>EISCONN</code>	il socket è già connesso, ma si è specificato un destinatario.
<code>EMSGSIZE</code>	il tipo di socket richiede l'invio dei dati in un blocco unico, ma la dimensione del messaggio lo rende impossibile.
<code>ENOBUFS</code>	la coda di uscita dell'interfaccia è già piena (di norma Linux non usa questo messaggio ma scarta silenziosamente i pacchetti).
<code>ENOTCONN</code>	il socket non è connesso e non si è specificata una destinazione.
<code>EOPNOTSUPP</code>	il valore di <code>flag</code> non è appropriato per il tipo di socket usato.
<code>EPIPE</code>	il capo locale della connessione è stato chiuso, si riceverà anche un segnale di <code>SIGPIPE</code> , a meno di non aver impostato <code>MSG_NOSIGNAL</code> in <code>flags</code> .

ed anche `EFAULT`, `EBADF`, `EINVAL`, `EINTR`, `ENOMEM`, `ENOTSOCK` 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).

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

---

<sup>2</sup>il prototipo illustrato è quello utilizzato dalle *glibc*, che seguono 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 nelle *libc4*, ma `size_t` nelle *libc5*; infine l'argomento `tolen` era `int` nei vari *BSD4.\** nelle *libc4* e nelle *libc5*.

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. ??, 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 prototipo<sup>3</sup> 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 ad 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 con i puntatori alle variabili dove la struttura contenente quest'ultimo e la relativa lunghezza saranno scritti (si noti che `fromlen` è un valore 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 connessione<sup>4</sup> o come una cessazione delle comunicazioni.

<sup>3</sup>il prototipo è quello delle *glibc* che seguono le *Single Unix Specification*, i vari *BSD4.\**, le *libc4* e le *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.\**, le *libc4* e le *libc5*.

<sup>4</sup>dato 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.

### 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.

---

```

1 int main(int argc, char *argv[])
2 {
3     int sock;
4     int 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.

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 dal parametro passato a riga di comando, convertendolo con `inet_nton`. 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 lettura<sup>5</sup> 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.

Se pertanto si è avuto cura di attivare il server del servizio *daytime*<sup>6</sup> 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

<sup>5</sup>si 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.

<sup>6</sup>di norma questo è un servizio standard fornito dal *superdemone inetd*, per cui basta abilitarlo nel file di configurazione di quest'ultimo, avendo cura di predisporre il servizio su UDP.

è quello di ricevere un pacchetto di richiesta ed inviare in risposta un pacchetto contenente una stringa con la data corrente.

---

```

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.

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 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*.

---

```

1 void ClientEcho(FILE * filein, int socket, struct sockaddr_in *serv_addr);
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_addr;
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_addr, 0, sizeof(serv_addr)); /* clear server address */
20    serv_addr.sin_family = AF_INET; /* address type is INET */
21    serv_addr.sin_port = htons(7); /* echo port is 7 */
22    /* build address using inet_pton */
23    if ((inet_pton(AF_INET, argv[optind], &serv_addr.sin_addr)) <= 0) {
24        perror("Address creation error");
25        return 1;
26    }
27    /* do read/write operations */
28    ClientEcho(stdin, sock, &serv_addr);
29    /* normal exit */
30    return 0;
31 }
```

---

*Figura 17.4:* Sezione principale della prima versione client per il servizio *echo* su UDP.

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 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

---

```

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(recvbuff), 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.

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,<sup>7</sup> 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 [11] 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

---

<sup>7</sup>questo 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.

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 pacchetti, che potranno essere inviati e ricevuti usando le normali funzioni `read` e `write`.<sup>8</sup>

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.

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 parametro e sostituendo le chiamate a `sendto` e `recvfrom` con chiamate a `read` e `write` come illustrato dal nuovo codice riportato in fig. 17.7.

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.

---

<sup>8</sup>in 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.

---

```

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.

## 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.

## 17.3 Altri socket

Tratteremo in questa sezione gli altri tipi particolari di socket supportati da Linux, come i *raw socket*, con i quali si possono forgiare direttamente i pacchetti a tutti i livelli dello stack dei protocolli, o i socket *netlink* che definiscono una interfaccia di comunicazione con il kernel.

---

```

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.

# Capitolo 18

## Socket avanzati

Esamineremo in questo capitolo le funzionalità più evolute della gestione dei socket TCP, come l'uso del I/O multiplexing (trattato in sez. 11.1) con i socket, l'uso delle opzioni dei socket e la gestione dei dati urgenti e *out-of-band*.

### 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 I dati *out-of-band*

Una caratteristica speciale dei socket TCP è quella della presenza dei cosiddetti dati *out-of-band* ...

### 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



# **Parte III**

# **Appendici**



# Appendice A

## Il livello di rete

In questa appendice prenderemo in esame i vari protocolli disponibili a livello di rete.<sup>1</sup> 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: l'*Internet Protocol IP* (che più propriamente si dovrebbe chiamare IPv4) ed la sua nuova versione denominata IPv6.

### A.1 Il protocollo IP

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, etc.).

#### A.1.1 Introduzione

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 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.

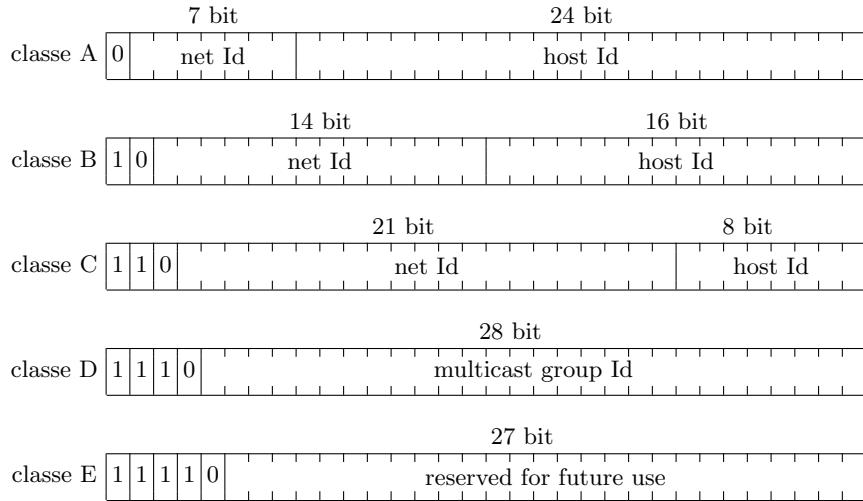
Per effettuare la comunicazione e l'instradamento dei pacchetti fra le varie reti di cui è composta Internet IPv4 organizza gli indirizzi in una gerarchia a due livelli, in cui una parte dei 32 bit dell'indirizzo indica il numero di rete, e un'altra l'host al suo interno. Il numero di rete serve ai router 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.

---

<sup>1</sup>per la spiegazione della suddivisione in livelli dei protocolli di rete, si faccia riferimento a quanto illustrato in sez. 13.2.

Per venire incontro alle diverse esigenze gli indirizzi di rete sono stati originariamente organizzati in *classi*, (rappresentate in tab. A.1), per consentire dispiegamenti di reti di dimensioni diverse.

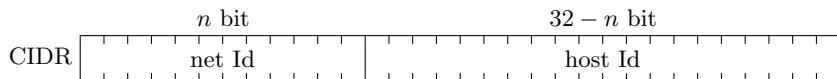


**Tabella A.1:** Le classi di indirizzi secondo IPv4.

Le classi usate per il dispiegamento delle reti sono le prime tre; la classe D è destinata al (non molto usato) *multicast* mentre la classe E è riservata per usi sperimentali e non viene impiegata.

Come si può notare però la suddivisione riportata in tab. A.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, 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 processo da parte di questi ultimi ed inefficienza nel trasporto.



**Tabella A.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 destinati a indicare l'host finale può essere piazzato in qualunque punto (vedi tab. A.2), permettendo di accoppare 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.

## A.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, etc.).

### A.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

### A.2.2 Principali caratteristiche di IPv6

Per rispondere alle esigenze descritte in sez. A.2.1 IPv6 nasce come evoluzione di IPv4, mantendone 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.

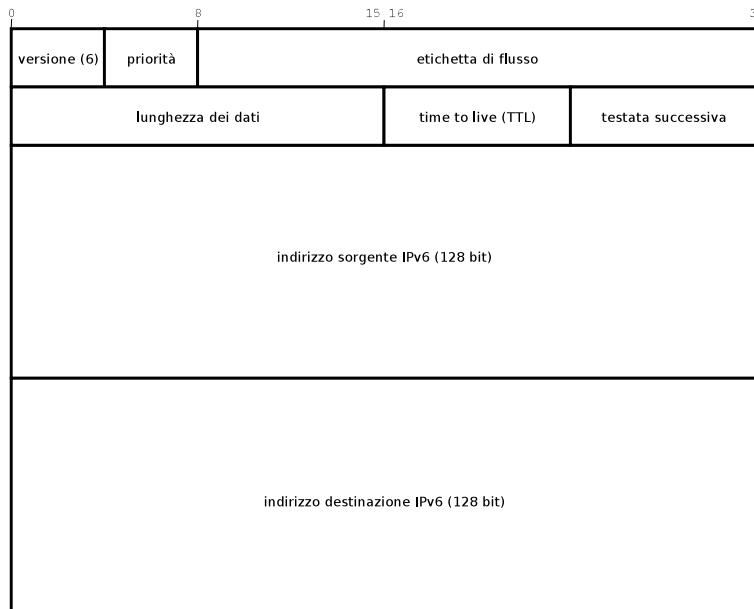
I cambiamenti apportati sono comunque notevoli e possono essere riassunti a grandi linee nei seguenti punti:

- 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 autoconfigurazione 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 riprocessamento della 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”)

### A.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. A.1 è riportato il formato dell'intestazione di IPv6 da confrontare con quella di IPv4 in fig. A.2. La spiegazione del significato dei vari campi delle due intestazioni è riportato rispettivamente in tab. A.3 e tab. A.4



**Figura A.1:** L'intestazione o *header* di IPv6.

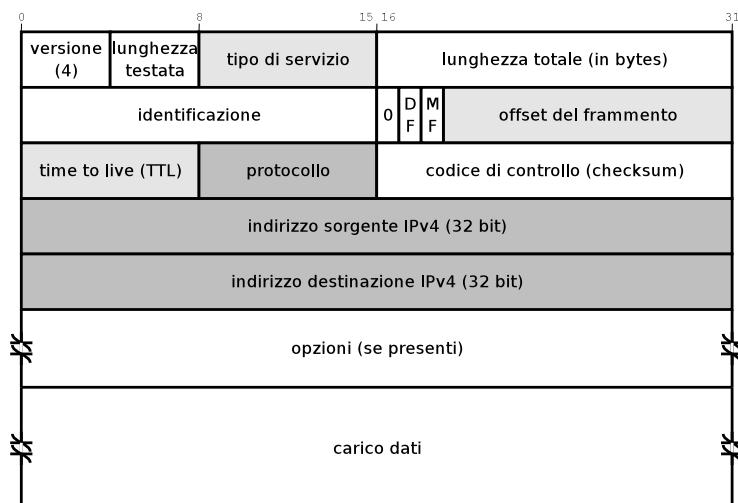
Come si può notare l'intestazione di IPv6 diventa di dimensione fissa, pari a 40 byte, contro una dimensione (minima, in assenza di opzioni) di 20 byte per IPv4; un semplice raddoppio nonostante lo spazio destinato agli indirizzi sia quadruplicato, questo grazie a una notevole semplificazione che ha ridotto il numero dei campi da 12 a 8.

Nome	Lunghezza	Significato
<i>version</i>	4 bit	<i>versione</i> , nel caso specifico vale sempre 6
<i>priority</i>	4 bit	<i>priorità</i> , vedi sez. A.2.15
<i>flow label</i>	24 bit	<i>etichetta di flusso</i> , vedi sez. A.2.14
<i>payload length</i>	16 bit	<i>lunghezza del carico</i> , cioè del corpo dei dati che segue l'intestazione, in byte.
<i>next header</i>	8 bit	<i>intestazione successiva</i> , identifica il tipo di pacchetto che segue l'intestazione di IPv6, usa gli stessi valori del campo protocollo nell'intestazione di IPv4
<i>hop limit</i>	8 bit	<i>limite di salti</i> , stesso significato del <i>time to live</i> nell'intestazione di IPv4, è decrementato di uno ogni volta che un nodo ritrasmette il pacchetto, se arriva a zero il pacchetto viene scartato
<i>source IP</i>	128 bit	<i>indirizzo di origine</i>
<i>destination IP</i>	128 bit	<i>indirizzo di destinazione</i>

**Tabella A.3:** Legenda per il significato dei campi dell'intestazione di IPv6

Abbiamo già anticipato in sez. A.2.2 uno dei criteri principali nella progettazione di IPv6 è stato quello di ridurre al minimo il tempo di processamento dei pacchetti da parte dei router, un confronto con l'intestazione di IPv4 (vedi fig. A.2) 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. A.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 il processo 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 molti tempo di riprocessamento 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. A.2.13) che permette di identificare i pacchetti appartenenti a un “*flusso*” di dati per i quali si può provvedere un trattamento speciale.



**Figura A.2:** L'intestazione o *header* di IPv4.

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:

Nome	Bit	Significato
<i>version</i>	4	<i>versione</i> , nel caso specifico vale sempre 4
<i>head length</i>	4	<i>lunghezza dell'intestazione</i> , in multipli di 32 bit
<i>type of service</i>	8	<i>tipo di servizio</i> , consiste in: 3 bit di precedenza, correntemente ignorati; un bit non usato a 0; 4 bit che identificano il tipo di servizio richiesto, uno solo dei quali può essere 1
<i>total length</i>	16	<i>lunghezza totale</i> , indica la dimensione del pacchetto IP in byte
<i>identification</i>	16	<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
<i>flag</i>	3	<i>flag</i> bit di frammentazione, uno indica se un pacchetto è frammentato, un'altro se ci sono ulteriori frammenti, e un'altro se il pacchetto non può essere frammentato.
<i>fragmentation offset</i>	13	<i>offset di frammento</i> , indica la posizione del frammento rispetto al pacchetto originale
<i>time to live</i>	16	<i>tempo di vita</i> , ha lo stesso significato di <i>hop limit</i> , vedi tab. A.3
<i>protocol</i>	8	<i>protocollo</i> identifica il tipo di pacchetto che segue l'intestazione di IPv4
<i>header checksum</i>	16	<i>checksum di intestazione</i> , somma di controllo per l'intestazione
<i>source IP</i>	32	<i>indirizzo di origine</i>
<i>destination IP</i>	32	<i>indirizzo di destinazione</i>

**Tabella A.4:** Legenda per il significato dei campi dell'intestazione di IPv4

- il broadcasting non è previsto in IPv6, le applicazioni che lo usano devono essere reimplementate usando il multicasting (vedi sez. A.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. A.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).

#### A.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.

#### A.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 esadecimalesi di 4 cifre (cioè a gruppi di 16 bit) usando i due punti come separatore; cioè qualcosa del tipo 5f1b:df00:ce3e:e200:0020:0800:2078:e3e3.

Visto che la notazione resta comunque piuttosto pesante esistono alcune abbreviazioni; si può evitare di scrivere gli zeri iniziali per cui si può scrivere 1080:0:0:0:8:800: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
geographic-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
multicast	1111 1111	1/256

**Tabella A.5:** Classificazione degli indirizzi IPv6 a seconda dei bit più significativi

### A.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. A.5 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. A.5 in quanto allocati al di fuori dello spazio di allocazione degli indirizzi unicast.

### A.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. A.6.

3	5 bit	$n$ bit	$56 - n$ bit	64 bit
010	Registry Id	Provider Id	Subscriber Id	Intra-Subscriber

**Tabella A.6:** 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 A.7:** 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. A.8 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.

64 bit	16 bit	48 bit
Subscriber Prefix	Subnet Id	Interface Id

**Tabella A.8:** Formato del campo *Intra-subscriber* per un indirizzo unicast *provider-based*.

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 MAC-address, a questo scopo si possono usare le capacità di autoconfigurazione 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, allostando 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. A.9.

3	5 bit	n bit	m bit	56-n-m bit	64 bit
3	Reg.	Naz.	Prov.	Subscr.	Intra-Subscriber

**Tabella A.9:** Formato di un indirizzo unicast *provider-based* che prevede un registro nazionale.

### A.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 autoconfigurazione prima di avere un indirizzo globale.

10	54 bit	64 bit
FE80	0000 . . . . 0000	Interface Id

**Tabella A.10:** 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. A.10, 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 A.2.19); un pacchetto che abbia tale indirizzo come sorgente o destinazione non deve venire ritrasmesso dai router.

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. A.11, 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.

10	38 bit	16 bit	64 bit
FEC0	0000 . . . 0000	Subnet Id	Interface Id

**Tabella A.11:** Formato di un indirizzo *site-local*.

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 con gli indirizzi *site-local* utilizzando un prefisso globale e la rinumerazione degli indirizzi delle singole macchine sarebbe automatica.

### A.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. A.12), 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).

**Tabella A.12:** Formato di un indirizzo IPV4 mappato su IPv6.

Un secondo tipo di indirizzi di compatibilità sono gli *IPv4 compatibili IPv6* (vedi tab. A.13) 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 encapsulato in IPv4.

**Tabella A.13:** 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.

### A.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. A.14:

8	4	4		112 bit
FF	flag	scop		Group Id

**Tabella A.14:** Formato di un indirizzo *multicast*.

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.
  - *scop* è un numero di quattro bit che indica il raggio di validità dell'indirizzo, i valori assegnati per ora sono riportati in tab. A.15.

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. A.16 sono già riservati per il funzionamento della rete.

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 A.15:** Possibili valori del campo *scop* di un indirizzo multicast.

Uso	Indirizzi riservati	Definizione
all-nodes	<b>FFxx:0:0:0:0:0:1</b>	RFC 1970
all-routers	<b>FFxx:0:0:0:0:0:2</b>	RFC 1970
all-rip-routers	<b>FFxx:0:0:0:0:0:9</b>	RFC 2080
all-cbt-routers	<b>FFxx:0:0:0:0:0:10</b>	
reserved	<b>FFxx:0:0:0:0:1:0</b>	IANA
link-name	<b>FFxx:0:0:0:0:1:1</b>	
all-dhcp-agents	<b>FFxx:0:0:0:0:1:2</b>	
all-dhcp-servers	<b>FFxx:0:0:0:0:1:3</b>	
all-dhcp-relays	<b>FFxx:0:0:0:0:1:4</b>	
solicited-nodes	<b>FFxx:0:0:0:1:0:0</b>	RFC 1970

**Tabella A.16:** Gruppi 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).

### A.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.

### A.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 processo, 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'indirizzamento 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. A.17.

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.

### A.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 temporale del flusso dei pacchetti.

Valore	Keyword	Tipo di protocollo
0		riservato
	HBH	Hop by Hop
1	ICMP	Internet Control Message (IPv4 o IPv6)
2	ICMP	Internet Group Management (IPv4)
3	GGP	Gateway-to-Gateway
4	IP	IP in IP (IPv4 encapsulation)
5	ST	Stream
6	TCP	Transmission 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 A.17:** Tipi di protocolli e intestazioni di estensione

#### A.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 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 FFFFFF in modo da rendere utilizzabile un qualunque sottoinsieme dei bit come chiavi di hash per i router.

#### A.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 A.18:** 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.

### A.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 da 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 stato 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.

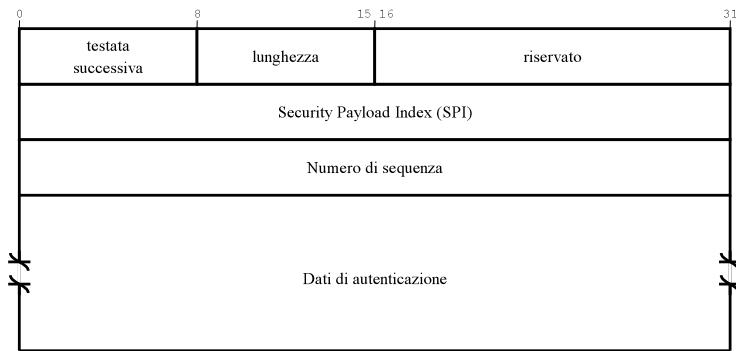
I pacchetti autenticati e crittografati portano un indice dei parametri di sicurezza (SPI, *Security Parameter Index*) che viene negoziato prima di ogni comunicazione ed è definito dalla stazione sorgente. Nel caso di multicast dovrà essere lo stesso per tutte le stazioni del gruppo.

### A.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. A.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.

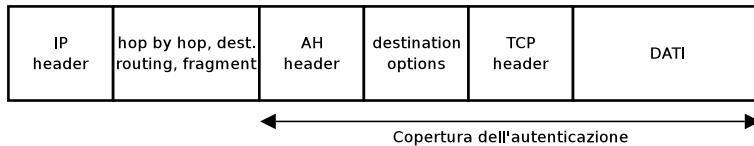
Complezano l'intestazione i dati di autenticazione che contengono un valore di controllo di integrità (ICV, *Integrity Check Value*), che deve essere di dimensione pari a un multiplo intero di 32 bit e può contenere un padding per allineare l'intestazione a 64 bit. Tutti gli algoritmi di autenticazione devono provvedere questa capacità.



**Figura A.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 eccezione fatta per la *Destination Option* che può comparire sia prima che dopo.



**Figura A.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.

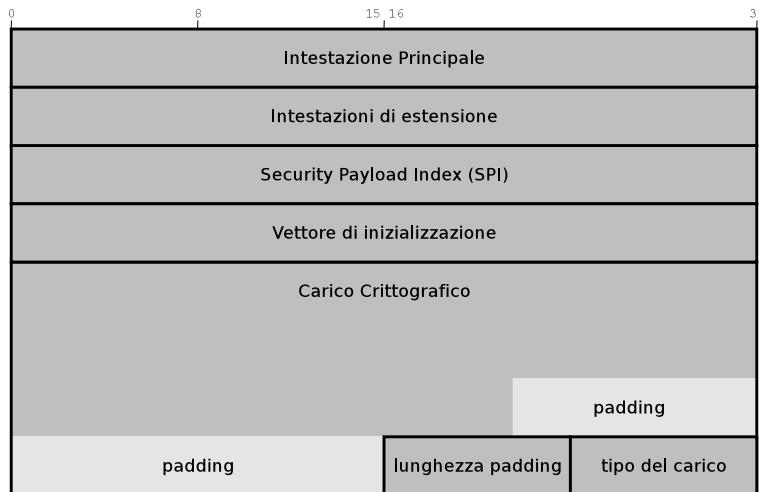
Il calcolo dei dati di autenticazione viene effettuato alla sorgente su una versione speciale del pacchetto in cui il numero di salti nell'intestazione principale è impostato a zero, così come le opzioni che possono essere modificate nella trasmissione, e l'intestazione di routing (se usata) è posta 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.

### A.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. ??, tutti i campi sono in chiaro fino al vettore di inizializzazione, il resto è crittografato.



**Figura A.5:** Schema di pacchetto crittografato.

### A.2.19 Autoconfigurazione

Una delle caratteristiche salienti di IPv6 è quella dell'autoconfigurazione, 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'autoconfigurazione 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 autoconfigurazione, descritte nelle seguenti sezioni. In ogni caso l'indirizzo link-local resta valido.

### A.2.20 Autoconfigurazione stateless

Questa è la forma più semplice di autoconfigurazione, 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 multicast *all-nodes* programmando la propria interfaccia per ricevere i messaggi dall'indirizzo multicast  $FF02::1$  (vedi sez. A.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 autocostruire

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.

### A.2.21 Autoconfigurazione stateful

Benché estremamente semplice l'autoconfigurazione 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 offre 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.



# Appendice B

## Il livello di trasporto

In questa appendice tratteremo i vari protocolli relativi al livello di trasporto.<sup>1</sup> 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.

### B.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.

#### B.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. B.1.

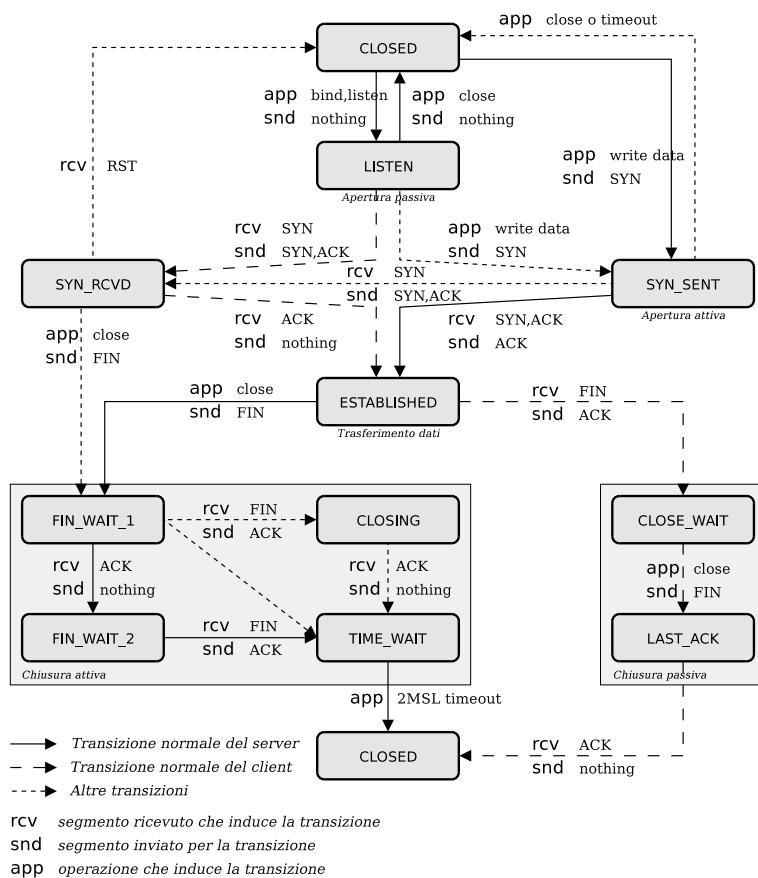
Il protocollo prevede l'esistenza di 11 diversi stati per una connessione ed un insieme di regole per le transizioni da uno stato all'altro basate sullo stato corrente, sull'operazione effettuata dall'applicazione o sul tipo di segmento ricevuto; i nomi degli stati mostrati in fig. B.1 sono gli stessi che vengono riportati del comando `netstat` nel campo *State*.

### B.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.

---

<sup>1</sup>al solito per la definizione dei livelli si faccia riferimento alle spiegazioni fornite in sez. 13.2.



**Figura B.1:** Il diagramma degli stati del TCP.

# Appendice C

## 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, qualora necessarie per il caso specifico, possono essere trovate nella descrizione del prototipo della funzione.

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. 8.5.2) possono essere usate per ottenere dei messaggi di errore più esplicativi.

### C.1 Gli errori dei file

In questa sezione sono raccolti i codici restituiti dalle funzioni di libreria attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei file.

**EPERM** *Operation not permitted.* L'operazione non è permessa: solo il proprietario del file o un processo con sufficienti privilegi può eseguire l'operazione.

**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.

**EIO** *Input/output error.* Errore di input/output: usato per riportare errori hardware in lettura/scrittura su un dispositivo.

**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.

**ENOEXEC** *Invalid executable file format.* Il file non ha un formato eseguibile, è un errore riscontrato dalle funzioni `exec`.

**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.

**EACCES** *Permission denied.* Permesso negato; l'accesso al file non è consentito: i permessi del file o della directory non consentono l'operazione.

**ELOOP** *Too many symbolic links encountered.* Ci sono troppi link simbolici nella risoluzione di un *pathname*.

**ENAMETOOLONG** *File name too long.* Si è indicato un *pathname* troppo lungo.

**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).

**EEXIST** *File exists.* Si è specificato un file esistente in un contesto in cui ha senso solo specificare un nuovo file.

**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.

**EXDEV** *Cross-device link.* Si è tentato di creare un link diretto che attraversa due filesystem differenti.

**ENODEV** *No such device.* Si è indicato un tipo di device sbagliato ad una funzione che ne richiede uno specifico.

**ENOTDIR** *Not a directory.* Si è specificato un file che non è una directory in una operazione che richiede una directory.

**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.

**EMFILE** *Too many open files.* Il processo corrente ha troppi file aperti e non può aprirne altri. Anche i descrittori duplicati vengono tenuti in conto<sup>1</sup>.

**ENFILE** *File table overflow.* Ci sono troppi file aperti nel sistema.

**ENOTTY** *Not a terminal.* Si è tentata una operazione di controllo relativa ad un terminale su un file che non lo è.

**ETXTBSY** *Text file busy.* Si è cercato di eseguire un file che è aperto in scrittura, o scrivere un file che è in esecuzione.

**EFBIG** *File too big.* Si è ecceduto il limite imposto dal sistema sulla dimensione massima che un file può avere.

**ENOSPC** *No space left on device.* la directory in cui si vuole creare il link non ha spazio per ulteriori voci.

**ESPIPE** *Invalid seek operation.*

**EROFS** *Read-only file system.* il file risiede su un filesystem read-only.

**EMLINK** *Too many links.* Ci sono troppi link al file (il numero massimo è specificato dalla variabile `LINK_MAX`, vedi sez. 8.1.1).

**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.

---

<sup>1</sup>Il numero massimo di file aperti è controllabile dal sistema, in Linux si può usare il comando `ulimit`.

**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.

**EUSERS** *Too many users.* Troppi utenti, il sistema delle quote rileva troppi utenti nel sistema.

**EDQUOT** *Quota exceeded.* Si è ecceduta la quota di disco dell'utente.

**ESTALE** *Stale NFS file handle.* Indica un problema interno a NFS causato da cambiamenti del filesystem del sistema remoto. Per recuperare questa condizione in genere è necessario smontare e rimontare il filesystem NFS.

**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.

**ENOLCK** *No locks available.* È usato dalle utilità per la gestione del file lock; non viene generato da un sistema GNU, ma può risultare da un'operazione su un server NFS di un altro sistema.

**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.

## C.2 Gli errori dei processi

In questa sezione sono raccolti i codici restituiti dalle funzioni di libreria attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei processi.

**ESRCH** *No process matches the specified process ID.* Non esiste un processo con il *pid* specificato.

**E2BIG** *Argument list too long.* Lista degli argomenti troppo lunga: è una condizione prevista da POSIX quando la lista degli argomenti passata ad una delle funzioni `exec` occupa troppa memoria, non può mai accadere in GNU/Linux.

**ECHILD** *There are no child processes.* Non esiste un processo figlio. Viene rilevato dalle funzioni per la gestione dei processi figli.

## C.3 Gli errori di rete

In questa sezione sono raccolti i codici restituiti dalle funzioni di libreria attinenti ad errori che riguardano operazioni specifiche relative alla gestione dei socket e delle connessioni di rete.

**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.

**EMSGSIZE** *Message too long.* Le dimensioni di un messaggio inviato su un socket sono eccedono la massima lunghezza supportata.

**EPROTOTYPE** *Protocol wrong type for socket.* Protocollo sbagliato per il socket. Il socket usato non supporta il protocollo di comunicazione richiesto.

**ENOPROTOOPT** *Protocol not available.* Protocollo non disponibile. Si è richiesta un'opzione per il socket non disponibile con il protocollo usato.

**EPROTONOSUPPORT** *Protocol not supported.* Protocollo non supportato. Il tipo di socket non supporta il protocollo richiesto (un probabile errore nella specificazione del protocollo).

**ESOCKTNOSUPPORT** *Socket type not supported.* Socket non supportato. Il tipo di socket scelto non è supportato.

**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.

**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.

**EADDRINUSE** *Address already in use.* L'indirizzo del socket richiesto è già utilizzato (ad esempio si è richiesto il `bind` per 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).

**ENETDOWN** *Network is down.* L'operazione sul socket è fallita perché la rete è sconnessa.

**ENETUNREACH** *Network is unreachable.* L'operazione è fallita perché l'indirizzo richiesto è irraggiungibile (ad esempio la sottorete della stazione remota è irraggiungibile).

**ENETRESET** *Network dropped connection because of reset.* Una connessione è stata cancellata perché l'host remoto è caduto.

**ECONNABORTED** *Software caused connection abort.* Una connessione è stata abortita localmente.

**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.

**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.

**EISCONN** *Transport endpoint is already connected.* Si è tentato di connettere un socket che è già connesso.

**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**.

**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.

**ESHUTDOWN** *Cannot send after transport endpoint shutdown.* Il socket su cui si cerca di inviare dei dati ha avuto uno shutdown.

**ETOOMANYREFS** *Too many references: cannot splice.* La glibc dice ???

**ETIMEDOUT** *Connection timed out.* Un'operazione sul socket non ha avuto risposta entro il periodo di timeout.

**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).

**EHOSTDOWN** *Host is down.* L'host remoto di una connessione è giù.

**EHOSTUNREACH** *No route to host.* L'host remoto di una connessione non è raggiungibile.

## C.4 Errori generici

In questa sezione sono raccolti i codici restituiti dalle funzioni di libreria attinenti ad errori generici, si trovano qui tutti i codici di errore non specificati nelle sezioni precedenti.

**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. In questo caso è necessario ripetere la chiamata alla funzione.

**ENOMEM** *No memory available.* Il kernel non è in grado di allocare ulteriore memoria per completare l'operazione richiesta.

**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 parametro è fuori dello spazio di indirizzi del processo, in genere questa situazione provoca l'emissione di un segnale di *segment violation* (**SIGSEGV**).

**EINVAL** *Invalid argument.* Errore utilizzato per segnalare vari tipi di problemi dovuti all'aver passato un argomento sbagliato ad una funzione di libreria.

**EDOM** *Domain error.* È usato dalle funzioni matematiche quando il valore di un argomento è al di fuori dell'intervallo in cui sono definite.

**ERANGE** *Range error.* È usato dalle funzioni matematiche quando il risultato non è rappresentabile a causa di un overflow o di un underflow.

**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.

**EWOULDBLOCK** *Operation would block.* In Linux è identico a **EAGAIN**.

**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 dovranno 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**.

**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.

**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 i parametri 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.

**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.

**EBADMSG** *Not a data message.*

**EIDRM** *Identifier removed.* Indica che l'oggetto del *SysV IPC* cui si fa riferimento è stato cancellato.

**EMULTIHOP** *Multihop attempted.*

**ENODATA** *No data available.*

**ENOLINK** *Link has been severed.*

**ENOMSG** *No message of desired type.*

**ENOSR** *Out of streams resources.*

**ENOSTR** *Device not a stream.*

**EOVERFLOW** *Value too large for defined data type.*

**EPROTO** *Protocol error.*

**ETIME** *Timer expired.*

## C.5 Errori del kernel

In questa sezione sono raccolti i codici di errore interni del kernel. Non sono usati dalle funzioni di libreria, ma vengono riportati da alcune system call (TODO verificare i dettagli, eventualmente cassare).

**ERESTART** *Interrupted system call should be restarted.*

**ECHRNG** *Channel number out of range.*

**EL2NSYNC** *Level 2 not synchronized.*

**EL3HLT** *Level 3 halted.*

**EL3RST** *Level 3 reset.*

**ELNRNG** *Link number out of range.*

**EUNATCH** *Protocol driver not attached.*

**ENOCSI** *No CSI structure available.*

**EL2HLT** *Level 2 halted.*

**EBADE** *Invalid exchange.*

**EBADR** *Invalid request descriptor.*

**EXFULL** *Exchange full.*

**ENOANO** *No anode.*

**EBADRQC** *Invalid request code.*

**EBADSLT** *Invalid slot.*

**EDEADLOCK** Identico a EDEADLK.

**EBFONT** *Bad font file format.*

**ENONET** *Machine is not on the network.*

**ENOPKG** *Package not installed.*

**EADV** *Advertise error.*

**ESRMNT** *Srmount error.*

**ECOMM** *Communication error on send.*

**EDOTDOT** *RFS specific error.*

**ENOTUNIQ** *Name not unique on network.*

**EBADFD** *File descriptor in bad state.*

**EREMCHG** *Remote address changed.*

**ELIBACC** *Can not access a needed shared library.*

**ELIBBAD** *Accessing a corrupted shared library.*

**ELIBSCN** *.lib section in a.out corrupted.*

**ELIBMAX** *Attempting to link in too many shared libraries.*

**ELIBEXEC** *Cannot exec a shared library directly.*

**ESTRPIPE** *Streams pipe error.*

**EUCLEAN** *Structure needs cleaning.*

**ENAVAIL** *No XENIX semaphores available.*

**EISNAM** *Is a named type file.*

**EREMOTEIO** *Remote I/O error.*

**ENOMEDIUM** *No medium found.*

**EMEDIUMTYPE** *Wrong medium type.*

## Appendice D

# 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 molteplici 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 ed il grande lavoro svolto per produrre una versione della guida in un HTML piacevole ed accurato.

Infine, ultimo, ma primo per importanza, voglio ringraziare il Firenze Linux User Group (FLUG), di cui mi prego di fare parte, che ha messo a disposizione il repository CVS, lo spazio web e tutto quanto è necessario alla pubblicazione della guida.



## Appendice E

# GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.  
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA  
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 written 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. Secondarily, 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.

## E.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

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. (For example, 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.

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 “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and 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 has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, L<sup>A</sup>T<sub>E</sub>X input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, 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 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.

## E.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.

## E.3 Copying in Quantity

If you publish printed copies 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 publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. 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.

## E.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:

- 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.
- 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 less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- 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.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and 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.

- 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.
- In any section entitled “Acknowledgements” or “Dedications”, preserve the section’s title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- 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.
- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

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.

## E.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.

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.”

## E.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.

## E.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, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they 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 quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

## E.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 provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

## E.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## E.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.

# Indice analitico

\_BSD\_SOURCE (macro), 11, 90, 151, 181, 225  
\_CHILD\_MAX (costante), 158  
\_DIRENT\_HAVE\_D\_NAMLEN (macro), 90  
\_DIRENT\_HAVE\_D\_OFF (macro), 91  
\_DIRENT\_HAVE\_D\_RECLEN (macro), 91  
\_DIRENT\_HAVE\_D\_TYPE (macro), 91  
\_GNU\_SOURCE (macro), 12, 119, 137, 143, 147,  
    151, 160, 198, 252  
\_IOFBF (costante), 151  
\_IOLBF (costante), 151  
\_IONBF (costante), 151  
\_ISOC99\_SOURCE (macro), 12  
\_LARGEFILE\_SOURCE (macro), 12  
\_OPEN\_MAX (costante), 158  
\_PATH\_UTMP (costante), 169  
\_PATH\_WTMP (costante), 169  
\_POSIX\_AIO\_LISTIO\_MAX (costante), 157  
\_POSIX\_AIO\_MAX (costante), 157  
\_POSIX\_ARG\_MAX (costante), 157  
\_POSIX\_ASYNCHRONOUS\_IO (macro), 256  
\_POSIX\_CHILD\_MAX (costante), 157  
\_POSIX\_C\_SOURCE (macro), 11, 12  
\_POSIX\_JOB\_CONTROL (macro), 157, 158, 223  
\_POSIX\_LINK\_MAX (costante), 159  
\_POSIX\_MAX\_CANON (costante), 159  
\_POSIX\_MAX\_INPUT (costante), 159  
\_POSIX\_MEMLOCK\_RANGE (macro), 24  
\_POSIX\_NAME\_MAX (costante), 159  
\_POSIX\_NGROUPS\_MAX (costante), 157  
\_POSIX\_OPEN\_MAX (costante), 157  
\_POSIX\_PATH\_MAX (costante), 159  
\_POSIX\_PIPE\_BUF (costante), 159  
\_POSIX\_PRIORITIZED\_IO (macro), 257  
\_POSIX\_PRIORITY\_SCHEDULING (macro), 67,  
    257  
\_POSIX\_SAVED\_IDS (macro), 55, 157, 158  
\_POSIX\_SIGQUEUE\_MAX (costante), 221  
\_POSIX\_SOURCE (macro), 11, 12, 61  
\_POSIX\_SSIZE\_MAX (costante), 157  
\_POSIX\_STREAM\_MAX (costante), 157  
\_POSIX\_THREAD\_SAFE\_FUNCTIONS (macro), 152  
\_POSIX\_TZNAME\_MAX (costante), 157  
\_POSIX\_VERSION (costante), 157, 158  
\_REENTRANT (macro), 69  
\_SC\_AVPHYS\_PAGES (costante), 175  
\_SC\_NPROCESSORS\_CONF (costante), 175  
\_SC\_NPROCESSORS\_ONLN (costante), 175  
\_SC\_PHYS\_PAGES (costante), 175  
\_SVID\_SOURCE (macro), 12, 90, 151  
\_SYS\_NMLN (costante), 161  
\_THREAD\_SAFE (macro), 69  
\_USE\_BSD (macro), 51  
\_UTSNAME\_DOMAIN\_LENGTH (costante), 160  
\_UTSNAME\_LENGTH (costante), 160  
\_XOPEN\_SOURCE (macro), 12, 201, 225, 252,  
    254  
\_XOPEN\_SOURCE\_EXTENDED (macro), 12, 225  
\_\_clone (funzione), 46  
\_\_fbuflen (funzione)  
    definizione di, 151  
\_\_flbf (funzione)  
    definizione di, 151  
\_\_freadable (funzione)  
    definizione di, 150  
\_\_freading (funzione)  
    definizione di, 150  
\_\_fsetlocking (funzione)  
    definizione di, 153  
\_\_fwritable (funzione)  
    definizione di, 150  
\_\_fwriteing (funzione)  
    definizione di, 150  
\_\_va\_copy (macro), 32  
\_exit (funzione), 14–16, 46, 47, 50, 228, 231  
    definizione di, 15  
\_flushlbf (funzione)  
    definizione di, 152  
deadlock, 270, 272, 286  
pathname, 84, 101, 160, 163  
race condition, 119, 328  
scheduler, 43  
self-pipe trick, 252  
three way handshake, 382  
Posix IPC names, 331

*Virtual File System*, 72, 74, 76  
*buffer overflow*, 142, 145, 147  
*close-on-exec*, 45, 53, 127, 129  
*copy on write*, 25, 41, 46, 264  
*deadlock*, 68–69, 209, 217, 249, 272, 507  
*deep copy*, 441, 452  
*endianess*, 366–368, 381  
*file descriptor set*, 250–251  
*memory leak*, 21, 22, 144, 147  
*memory locking*, 24–25, 318  
*page fault*, 17, 18, 25, 64, 172  
*pathname*, 40, 53, 71–504
 

- assoluto, 72, 113, 332
- relativo, 72, 113

*polling*, 250, 303, 328, 329  
*preemptive scheduling*, 3  
*race condition*, 44, 67–69, 96–98, 121, 126, 190, 192, 208–210, 216, 217, 226, 252, 267, 281  
*resolver*, 431–440, 449  
*scheduler*, 3, 38, 39, 61, 63, 65, 175, 191, 192, 207, 313  
*signal mask*, 215–217  
*signal set*, 211–212, 250  
*three way handshake*, 371–372, 382–385, 407, 409  
*value result argument*, 31, 59, 90, 144, 147, 442, 468

**abort** (funzione), 14, 21, 46, 194, 195, 203, 205
 

- definizione di, 205

**accept** (funzione), 371, 372, 383–387, 392, 394, 401, 403–408, 415, 416, 425, 427, 465, 467
 

- definizione di, 385, 408

**access** (funzione), 85, 102, 108, 109
 

- definizione di, 108

**addrinfo** (struttura dati), 447–452, 454
 

- definizione di, 447

**ADJ\_ESTERROR** (costante), 180  
**ADJ\_FREQUENCY** (costante), 180  
**ADJ\_MAXERROR** (costante), 180  
**ADJ\_OFFSET** (costante), 180  
**ADJ\_OFFSET\_SINGLESHOT** (costante), 180  
**ADJ\_STATUS** (costante), 180  
**ADJ\_TICK** (costante), 180  
**ADJ\_TIMECONST** (costante), 180  
**adjtime** (funzione), 180
 

- definizione di, 179

**adjtimex** (funzione), 181

**definizione di**, 180  
**AF\_APPLETALK** (costante), 364  
**AF\_INET** (costante), 362, 369, 389, 439, 443, 461, 470  
**AF\_INET6** (costante), 363, 369, 439, 443  
**AF\_PACKET** (costante), 366  
**AF\_UNIX** (costante), 292, 363, 380  
**AI\_ADDRCONFIG** (costante), 443, 448  
**AI\_ALL** (costante), 443, 448  
**AI\_CANONNAME** (costante), 448, 449  
**AI\_DEFAULT** (costante), 443  
**AI\_NUMERICHOST** (costante), 448  
**AI\_PASSIVE** (costante), 448, 456  
**AI\_V4MAPPED** (costante), 443, 448  
**AI\_V4MAPPED** (costante), 443  
**AIO\_ALLDONE** (costante), 259  
**aio\_cancel** (funzione), 259
 

- definizione di, 259

**AIO\_CANCELED** (costante), 259  
**aio\_error** (funzione), 258, 259
 

- definizione di, 258

**aio\_fsync** (funzione), 258  
**AIO\_LISTIO\_MAX** (costante), 260  
**AIO\_NOTCANCELED** (costante), 259  
**aio\_read** (funzione), 258
 

- definizione di, 257

**aio\_return** (funzione), 259
 

- definizione di, 258

**aio\_suspend** (funzione)
 

- definizione di, 259

**aio\_write** (funzione), 258
 

- definizione di, 257

**aiocb** (struttura dati), 256–258, 260
 

- definizione di, 256

**alarm** (funzione), 194, 196, 203–207, 209, 210, 216
 

- definizione di, 203

**alloca** (funzione), 22
 

- definizione di, 22

**alphasort** (funzione), 93
 

- definizione di, 92

**ARG\_MAX** (costante), 157, 158  
**asctime** (funzione), 181, 182  
**asprintf** (funzione)
 

- definizione di, 147

**AT\_ANYNET** (costante), 364  
**AT\_ANYNODE** (costante), 364  
**at\_exit** (funzione), 205  
**ATADDR\_BCAST** (costante), 364  
**atexit** (funzione), 14, 15

definizione di, 15  
**atohs** (funzione), 394  
**ATPROTO\_DDP** (funzione), 364

**bind** (funzione), 362, 365, 366, 371, 380, 381, 383, 387, 392, 398, 401, 448, 455, 456, 462, 463, 465, 466, 472, 506  
definizione di, 380

**BOOT\_TIME** (costante), 170  
**brk** (funzione), 22, 174, 319  
definizione di, 22

**BRKINT** (costante), 238  
**BSDLY** (costante), 239  
**BUFSIZ** (costante), 150, 151

**C\_ANY** (costante), 436  
**C\_CHAOS** (costante), 436  
**C\_CSNET** (costante), 436  
**C\_HS** (costante), 436  
**C\_IN** (costante), 436  
**caddr\_t** (tipo), 8  
**calloc** (funzione), 20  
definizione di, 20

**CAP\_NET\_ADMIN** (costante), 462  
**CAP\_NET\_BIND\_SERVICE** (costante), 364  
**CAP\_NET\_RAW** (costante), 365  
**CBAUD** (costante), 240  
**CBAUDEX** (costante), 240  
**cfgetispeed** (funzione)  
definizione di, 245  
**cfgetospeed** (funzione)  
definizione di, 245  
**cfree** (funzione), 20  
**cfsetispeed** (funzione)  
definizione di, 244  
**cfsetospeed** (funzione)  
definizione di, 244  
**char \*** (tipo), 442  
**CHAR\_BIT** (costante), 156  
**CHAR\_MAX** (costante), 156  
**CHAR\_MIN** (costante), 156  
**chdir** (funzione), 85, 94, 96, 230, 231, 323  
definizione di, 96  
**CHILD\_MAX** (costante), 157  
**chmod** (funzione), 85, 102, 109, 110, 230  
definizione di, 109  
**chown** (funzione), 85, 102, 111, 230  
definizione di, 111  
**chroot** (funzione), 72, 113, 114  
definizione di, 113  
**CIBAUD** (costante), 240

**clearenv** (funzione), 29  
definizione di, 30  
**clearerr** (funzione)  
definizione di, 138  
**clearerr\_unlocked** (funzione), 138  
**ClientEcho** (funzione), 410, 411, 416, 417, 421, 422, 473, 474, 478  
**CLK\_TCK** (costante), 158, 176  
**CLOCAL** (costante), 240  
**clock** (funzione)  
definizione di, 176  
**clock\_t** (tipo), 8, 176, 177  
**CLOCKS\_PER\_SEC** (costante), 176, 177  
**close** (funzione), 117, 120, 137, 334, 339, 340, 373, 374, 386, 387, 394, 419, 420, 468  
definizione di, 120  
**closedir** (funzione)  
definizione di, 92  
**ComputeValues** (funzione), 325  
**connect** (funzione), 371, 372, 381–383, 385, 387, 390, 401, 448, 454–456, 461, 466, 467, 473, 476, 508  
definizione di, 382  
**const** (direttiva), 30  
**CRDLY** (costante), 239  
**CREAD** (costante), 240  
**creat** (funzione), 85, 102, 126, 278  
definizione di, 119  
**CreateMutex** (funzione), 329  
**CreateShm** (funzione), 340  
**CRTSCTS** (costante), 240  
**CSIZE** (costante), 240  
**CSTOPB** (costante), 240  
**ctermid** (funzione)  
definizione di, 237  
**ctime** (funzione), 181, 390, 394

**daemon** (funzione), 289, 291, 303, 324, 326, 392, 398  
definizione di, 231  
**DEAD\_PROCESS** (costante), 170  
**DEFECCHO** (costante), 241  
**dentry** (struttura dati), 116  
**dev\_t** (tipo), 8  
**DIR** (tipo), 90  
**dirent** (struttura dati), 90–93, 95  
definizione di, 91  
**dirfd** (funzione), 94  
definizione di, 90  
**DirProp** (struttura dati), 322, 324, 325

- DirScan** (funzione), 94, 325  
**DoS**, 118, 119, 264  
**dprintf** (funzione), 147  
**drand48** (funzione), 11  
**DT\_BLK** (costante), 91  
**DT\_CHR** (costante), 91  
**DT\_DIR** (costante), 91  
**DT\_FIFO** (costante), 91  
**DT\_REG** (costante), 91  
**DT SOCK** (costante), 91  
**DT UNKNOWN** (costante), 91  
**DTTOIF** (macro), 91  
**dup** (funzione), 45, 127, 128, 136, 268, 269, 273  
  definizione di, 127  
**dup2** (funzione), 127, 128, 282  
  definizione di, 128  
  
**E2BIG** (errore), 51, 302, 311, 505  
**EACCES** (errore), 51, 52, 64, 87, 88, 95, 96, 101, 103, 108, 130, 163, 225, 263, 270, 271, 297, 298, 300, 302, 309, 311, 318, 319, 358, 380, 382, 503  
**EACCESS** (errore), 333  
**EADDRINUSE** (errore), 380, 462, 506  
**EADDRNOTAVAIL** (errore), 380, 506  
**EADV** (errore), 509  
**EAFNOSUPPORT** (errore), 292, 369, 382, 506  
**EAGAIN** (errore), 41, 121–123, 130, 174, 221, 222, 249, 257, 259–261, 263, 271, 277, 278, 300, 301, 311, 312, 333, 335, 336, 382, 385, 386, 461, 467, 468, 507, 508  
**EAI\_ADDRFAMILY** (costante), 449  
**EAI AGAIN** (costante), 449  
**EAI\_BADFLAGS** (costante), 449  
**EAI\_FAIL** (costante), 449  
**EAI\_FAMILY** (costante), 449  
**EAI\_MEMORY** (costante), 449  
**EAI\_NODATA** (costante), 449  
**EAI\_NONAME** (costante), 448, 449  
**EAI\_SERVICE** (costante), 449  
**EAI\_SOCKTYPE** (costante), 449  
**EAI\_SYSTEM** (costante), 448, 449  
**EALREADY** (errore), 382, 508  
**EBADE** (errore), 509  
**EBADF** (errore), 101, 120, 127–129, 250, 252, 253, 257, 258, 261, 263, 337, 380, 383, 385–387, 457, 458, 503  
**EBADFD** (errore), 509  
**EBADMSG** (errore), 508  
  
**EBADR** (errore), 509  
**EBADRQC** (errore), 509  
**EBADSLT** (errore), 509  
**EBFONT** (errore), 509  
**EBUSY** (errore), 84, 88, 163–165, 337, 504  
**ECANCELED** (errore), 259  
**ECHILD** (errore), 49, 505  
**ECHO** (costante), 241  
**ECHOCTL** (costante), 241  
**ECHOE** (costante), 240, 241  
**ECHOK** (costante), 240, 241  
**ECHOKE** (costante), 240, 241  
**ECHONL** (costante), 240, 241  
**ECHOPRT** (costante), 240, 241  
**ECHRNG** (errore), 509  
**ECOMM** (errore), 509  
**ECONNABORTED** (errore), 506  
**ECONNREFUSED** (errore), 382, 383, 468, 507  
**ECONNRESET** (errore), 408, 414, 419, 460, 467, 506  
**EDEADLK** (errore), 270, 272, 507, 509  
**EDEADLOCK** (errore), 509  
**EDESTADDRREQ** (errore), 467, 506  
**EDOM** (errore), 507  
**EDOTDOT** (errore), 509  
**EDQUOT** (errore), 505  
**EEXIST** (errore), 82, 85, 87, 88, 97, 98, 117, 118, 297, 298, 333, 504  
**EFAULT** (errore), 60, 212, 215, 216, 218, 266, 318, 386, 387, 457, 458, 507  
**EFBIG** (errore), 123, 504  
**EFTYPE** (errore), 505  
**EHOSTDOWN** (errore), 408, 507  
**EHOSTUNREACH** (errore), 408, 412, 413, 418, 507  
**EIDRM** (errore), 297, 300–302, 309, 311, 312, 318, 508  
**EILSEQ** (errore), 508  
**EINPROGRESS** (errore), 258, 382, 508  
**EINTR** (errore), 48, 49, 97, 120–123, 130, 200, 207, 222, 242, 250–253, 259–261, 270, 271, 300–302, 311, 312, 333, 385, 388, 403, 404, 406, 425, 507  
**EINVAL** (errore), 24, 51, 60, 63–66, 84, 86, 88, 89, 95, 98, 101, 108, 120, 123, 127, 129, 131, 161, 163, 173, 185, 202, 207, 212, 215, 216, 218, 221, 222, 225, 250, 252, 253, 257, 258, 260, 261, 263, 266, 284, 300, 302,

307, 309, 316, 318, 319, 333–336, 358, 362, 380, 457, 507  
**EIO** (errore), 227, 240, 503  
**EISCONN** (errore), 467, 468, 506  
**EISDIR** (errore), 82, 84, 117, 504  
**EISNAM** (errore), 510  
**EL2HLT** (errore), 509  
**EL2NSYNC** (errore), 509  
**EL3HLT** (errore), 509  
**EL3RST** (errore), 509  
**ELIBACC** (errore), 509  
**ELIBBAD** (errore), 51, 509  
**ELIBEXEC** (errore), 510  
**ELIBMAX** (errore), 510  
**ELIBSCN** (errore), 510  
**ELNRNG** (errore), 509  
**ELOOP** (errore), 87, 117, 504  
**EMEDIUMTYPE** (errore), 510  
**EMFILE** (errore), 127–129, 163, 174, 358, 504  
**EMLINK** (errore), 82, 87, 504  
**EMPTY** (costante), 170  
**EMSGSIZE** (errore), 335, 336, 467, 505  
**EMULTIHOP** (errore), 508  
**ENAMETOOLONG** (errore), 504  
**ENAVAIL** (errore), 510  
**endgrent** (funzione), 168  
**endhostent** (funzione), 446  
    definizione di, 442  
**endián** (funzione), 367  
**endnetent** (funzione), 446  
**endprotoent** (funzione), 446  
**endpwent** (funzione), 168  
**endservent** (funzione), 445, 446  
    definizione di, 445, 446  
**endutent** (funzione)  
    definizione di, 169  
**endutxent** (funzione), 171  
**ENETDOWN** (errore), 408, 506  
**ENETRESET** (errore), 506  
**ENETUNREACH** (errore), 382, 383, 408, 413, 506  
**ENFILE** (errore), 358, 504  
**ENOAFSUPPORT** (errore), 369  
**ENOANO** (errore), 509  
**ENOBUFS** (errore), 358, 385–387, 467, 506  
**ENOCSI** (errore), 509  
**ENODATA** (errore), 508  
**ENODEV** (errore), 117, 163, 263, 504  
**ENOENT** (errore), 51, 85, 103, 297, 298, 333, 503  
**ENOEXEC** (errore), 51, 503  
**ENOLCK** (errore), 270, 276, 505  
**ENOLINK** (errore), 508  
**ENOMEDIUM** (errore), 510  
**ENOMEM** (errore), 24, 41, 161, 218, 263, 307, 311, 316, 335, 385, 506, 507  
**ENOMSG** (errore), 302, 508  
**ENONET** (errore), 408, 509  
**ENOPKG** (errore), 509  
**ENOPROTOOPT** (errore), 408, 457, 458, 460, 505  
**ENOSPC** (errore), 87, 297, 307, 316, 369, 504  
**ENOSR** (errore), 508  
**ENOSTR** (errore), 508  
**ENOSYS** (errore), 67, 165, 227, 257, 259, 260, 508  
**ENOTBLK** (errore), 163, 504  
**ENOTCONN** (errore), 387, 419, 467, 468, 506  
**ENOTDIR** (errore), 84, 96, 117, 161, 504  
**ENOTEMPTY** (errore), 84, 88, 505  
**ENOTSOCK** (errore), 380, 383, 385–387, 419, 457, 458, 505  
**ENOTSUP** (errore), 508  
**ENOTTY** (errore), 131, 227, 236, 504  
**ENOTUNIQ** (errore), 509  
**ENXIO** (errore), 117, 163, 286, 503  
**EOPNOTSUPP** (errore), 261, 292, 383, 385, 408, 467, 506  
**EOVERFLOW** (errore), 318, 508  
**EPERM** (errore), 51, 56, 60, 63–65, 82, 85, 88, 109–111, 113, 161, 163, 164, 173, 178–180, 202, 218, 221, 225, 227, 300, 309, 318, 382, 385, 503  
**EPFNOSUPPORT** (errore), 506  
**EPIPE** (errore), 121, 123, 198, 281, 467, 504  
**EPROTO** (errore), 408, 508  
**EPROTONOSUPPORT** (errore), 292, 358, 506  
**EPROTOTYPE** (errore), 505  
**ERANGE** (errore), 95, 167, 185, 237, 309, 311, 312, 442, 507  
**EREMCHG** (errore), 509  
**EREMOTE** (errore), 505  
**EREMOTEIO** (errore), 510  
**ERESTART** (errore), 509  
**EROFS** (errore), 82, 85, 108, 109, 504  
**error** (funzione), 186, 187  
    definizione di, 186  
**error\_at\_line** (funzione)  
    definizione di, 187  
**ESHUTDOWN** (errore), 507

- E\_SOCKETNOSUPPORT** (errore), 506
- ESPIPE** (errore), 120, 284, 504
- ESRCH** (errore), 63–67, 202, 221, 225, 505
- ESRMNT** (errore), 509
- ESTALE** (errore), 505
- ESTRPIPE** (errore), 510
- ETH\_P\_ALL** (costante), 365
- ETIME** (errore), 508
- ETIMEDOUT** (errore), 335, 336, 382, 413, 414, 418, 460, 507
- ETOOMANYREFS** (errore), 507
- ETXTBSY** (errore), 51, 101, 117, 263, 264, 504
- EUCLEAN** (errore), 510
- EUNATCH** (errore), 509
- EUSERS** (errore), 505
- EWOULDBLOCK** (errore), 122, 268, 276, 385, 461, 507, 508
- EXDEV** (errore), 82, 84, 504
- exec** (funzione), 13, 16, 18, 19, 25, 28, 30, 40, 43, 46, 51–55, 58, 85, 102, 119, 127, 129, 157, 158, 173, 199, 219, 225, 229, 230, 265, 272, 293, 313, 320, 387, 503, 505
- exec1** (funzione), 31, 33, 52
- execle** (funzione), 52
- execlp** (funzione), 52
- execv** (funzione), 52
- execve** (funzione), 16, 46, 52
  - definizione di, 51
- execvp** (funzione), 52
- execve** (funzione), 230
- EXFULL** (errore), 509
- exit** (funzione), 14–16, 39, 46, 47, 50, 54, 186, 320, 394, 402
  - definizione di, 14
- EXIT\_FAILURE** (costante), 14
- EXIT\_SUCCESS** (costante), 14
- extern** (direttiva), 381
- F\_DUPFD** (costante), 128, 129
- F\_GETFD** (costante), 129
- F\_GETFL** (costante), 129, 131
- F\_GETLEASE** (costante), 130
- F\_GETLK** (costante), 129, 271, 329
- F\_GETOWN** (costante), 130, 255
- F\_GETSIG** (costante), 130
- F\_NOTIFY** (costante), 130
- F\_OK** (costante), 109
- f\_ops** (struttura dati), 76
- F\_RDLCK** (costante), 271, 331
- F\_SETFD** (costante), 129
- F\_SETFL** (costante), 129, 255
- F\_SETLEASE** (costante), 130
- F\_SETLK** (costante), 130, 271, 272, 275
- F\_SETLKW** (costante), 130, 271, 275
- F\_SETLKW** (funzione), 329
- F\_SETOWN** (costante), 130, 255
- F\_SETSIG** (costante), 130, 255
- F\_UNLCK** (costante), 271, 331
- F\_WRLCK** (costante), 271, 331
- fchdir** (funzione), 90, 94
  - definizione di, 96
- fchmod** (funzione), 102, 109, 110
  - definizione di, 109
- fchown** (funzione), 102, 111
  - definizione di, 111
- fclose** (funzione), 15, 136
  - definizione di, 137
- fcloseall** (funzione)
  - definizione di, 137
- fcntl** (funzione), 53, 119, 128, 129, 131, 132, 136, 149, 196, 255, 267, 268, 270–272, 276, 277, 329, 386, 463
  - definizione di, 129
- FD\_CLOEXEC** (costante), 125, 129, 339
- FD\_CLR** (macro), 421
- FD\_ISSET** (costante), 251
- FD\_ISSET** (macro), 418
- fd\_set** (tipo), 250, 251
- FD\_SETSIZE** (costante), 250, 251
- FD\_SETSIZE** (macro), 423
- FD\_ZERO** (macro), 251
- fdatasync** (funzione), 259
  - definizione di, 127
- fdopen** (funzione), 136, 137
  - definizione di, 136
- feof** (funzione), 140
  - definizione di, 138
- ferror** (funzione), 140
  - definizione di, 138
- FFDLY** (costante), 239
- fflush** (funzione), 136, 137
  - definizione di, 151
- fflush\_unlocked** (funzione), 151
- fgetc** (funzione), 140
  - definizione di, 140
- fgetgrent** (funzione), 168
- fgetgrent\_r** (funzione), 168
- fgetpos** (funzione)
  - definizione di, 149
- fgetpwent** (funzione), 168

**fgetpwent\_r** (funzione), 168  
**fgets** (funzione), 142–144, 397, 401, 402, 410, 418, 421  
definizione di, 142  
**fgetwc** (funzione)  
definizione di, 141  
**fgetws** (funzione)  
definizione di, 143  
**file**  
descriptor, 74, 115–116  
di lock, 119, 126, 193, 327–329  
di dispositivo, 71–73, 76, 83, 88, 148, 200, 249  
locking, 74, 120, 125, 129, 131, 267–278, 316, 329, 330  
stream, 74, 133  
**file** (struttura dati), 76, 116, 118–120, 124, 125  
**file4** (funzione), 93  
**file\_lock** (struttura dati), 269, 272  
**file\_struct** (struttura dati), 117, 124, 125, 127  
**fileno** (funzione), 417  
definizione di, 149  
**files\_struct** (struttura dati), 115  
**FindMutex** (funzione), 329  
**FindShm** (funzione), 340  
**FIOASYNC** (costante), 132  
**FIOCLEX** (costante), 132  
**FIONBIO** (costante), 132  
**FIONCLEX** (costante), 132  
**FL\_FLOCK** (costante), 269  
**FL\_POSIX** (costante), 269, 272  
**flock** (funzione), 267–270, 275, 277  
definizione di, 268  
**flock** (struttura dati), 270, 271  
definizione di, 270  
**flockfile** (funzione)  
definizione di, 152  
**FLUSH0** (costante), 241  
**fmtmsg** (funzione), 11  
**fnctl** (funzione), 128  
**fopen** (funzione), 136, 137  
definizione di, 136  
**FOPEN\_MAX** (costante), 156  
**fork** (funzione), 39–46, 54, 58, 119, 124, 127, 173, 174, 199, 226, 229, 231, 265, 268–270, 272, 280, 281, 284, 313, 320, 394, 401, 508  
definizione di, 41  
**fpathconf** (funzione)  
definizione di, 160  
**fpos\_t** (tipo), 149  
**fprintf** (funzione)  
definizione di, 145  
**fpurge** (funzione)  
definizione di, 152  
**fputc** (funzione)  
definizione di, 141  
**fputs** (funzione), 143, 390, 397, 410  
definizione di, 143  
**fputws** (funzione)  
definizione di, 143  
**fread** (funzione), 139–141  
definizione di, 139  
**fread\_unlocked** (funzione)  
definizione di, 140  
**free** (funzione), 20–22, 144, 147  
definizione di, 20  
**freeaddrinfo** (funzione), 452  
definizione di, 451  
**freehostent** (funzione)  
definizione di, 444  
**freopen** (funzione), 134, 136  
definizione di, 136  
**fs\_struct** (struttura dati), 111, 113  
**fscanf** (funzione)  
definizione di, 147  
**fseek** (funzione), 137, 148, 149  
definizione di, 148  
**fseeko** (funzione), 149  
**FSETLOCKING\_BYCALLER** (costante), 153  
**FSETLOCKING\_INTERNAL** (costante), 153  
**FSETLOCKING\_QUERY** (costante), 153  
**fsetpos** (funzione), 137  
definizione di, 149  
**fstab** (struttura dati), 166  
**fstat** (funzione), 99, 127  
definizione di, 99  
**fstat** (struttura dati), 104  
**fstatfs** (funzione), 165  
definizione di, 165  
**fsync** (funzione), 126, 127, 152, 258, 259  
definizione di, 127  
**ftell** (funzione), 149  
definizione di, 149  
**ftello** (funzione), 149  
**ftok** (funzione), 326  
definizione di, 294  
**ftruncate** (funzione), 101, 102, 339, 340

**definizione di**, 101  
**ftrylockfile** (funzione)  
    definizione di, 152  
**FullRead** (funzione), 388, 389  
**FullWrite** (funzione), 388, 390, 394, 397,  
    400, 426  
**funlockfile** (funzione)  
    definizione di, 152  
**fwrite** (funzione), 140, 141  
    definizione di, 139  
**fwrite\_unlocked** (funzione)  
    definizione di, 140  
  
**gai\_strerror** (funzione), 448  
**geipnodebyname** (funzione), 452  
**get\_avphys\_pages** (funzione)  
    definizione di, 174  
**get\_phys\_pages** (funzione)  
    definizione di, 174  
**getaddrinfo** (funzione), 447–452, 454, 456  
    definizione di, 446  
**GETALL** (costante), 310  
**getc** (funzione), 140, 153  
    definizione di, 140  
**getchar** (funzione), 140  
    definizione di, 140  
**getcwd** (funzione), 96  
    definizione di, 95  
**getdelim** (funzione), 144  
    definizione di, 144  
**getegid** (funzione), 57  
    definizione di, 55  
**getenv** (funzione), 29, 30  
    definizione di, 29  
**geteuid** (funzione)  
    definizione di, 55  
**getgid** (funzione)  
    definizione di, 55  
**getgrent** (funzione), 168  
**getgrent\_r** (funzione), 168  
**getgrgid** (funzione)  
    definizione di, 167  
**getgrnam** (funzione)  
    definizione di, 167  
**getgrouplist** (funzione)  
    definizione di, 60  
**getgroups** (funzione), 59  
    definizione di, 60  
**gethostbyaddr** (funzione), 443, 444, 446  
    definizione di, 442  
  
**gethostbyname** (funzione), 435, 439–441, 443,  
    444, 446, 448, 452  
    definizione di, 438  
**gethostbyname2** (funzione), 441  
    definizione di, 439  
**gethostbyname2\_r** (funzione), 441  
    definizione di, 441  
**gethostbyname\_r** (funzione)  
    definizione di, 441  
**gethostent** (funzione), 446  
**getipnodebyaddr** (funzione), 443, 444, 446  
    definizione di, 443  
**getipnodebyname** (funzione), 443, 444, 446,  
    448  
    definizione di, 443  
**getitimer** (funzione), 205  
    definizione di, 205  
**getline** (funzione), 144  
    definizione di, 144  
**getloadavg** (funzione)  
    definizione di, 175  
**getnameinfo** (funzione), 446, 452  
    definizione di, 452  
**GETNCNT** (costante), 310  
**getnetbyaddr** (funzione), 444  
**getnetbyname** (funzione), 444  
**getnetent** (funzione), 446  
 **getopt** (funzione), 26, 27  
    definizione di, 26  
**getpagesize** (funzione)  
    definizione di, 174  
**getpeername** (funzione), 386, 387  
    definizione di, 387  
**getpgid** (funzione), 224  
    definizione di, 224  
**getpgrp** (funzione), 224  
    definizione di, 224  
**GETPID** (costante), 310  
**getpid** (funzione)  
    definizione di, 40  
**getppid** (funzione), 41  
    definizione di, 40  
**getpriority** (funzione), 64  
    definizione di, 63  
**getprotobyaddr** (funzione), 444  
**getprotobynumber** (funzione), 444  
**getprotoent** (funzione), 446  
**getpwent** (funzione), 168  
**getpwent\_r** (funzione), 168  
**getpwnam** (funzione), 230

definizione di, 166  
**getpwuid** (funzione)  
    definizione di, 166  
**getresgid** (funzione)  
    definizione di, 59  
**getresuid** (funzione)  
    definizione di, 59  
**getrlimit** (funzione), 174  
    definizione di, 173  
**getrusage** (funzione), 51  
    definizione di, 172  
**gets** (funzione), 142, 143, 411  
    definizione di, 142  
**getservbyaddr** (funzione), 444, 446  
    definizione di, 444  
**getservbyname** (funzione), 444–446  
    definizione di, 444  
**getservbyport** (funzione), 445, 446  
**getservent** (funzione), 445, 446  
    definizione di, 445  
**getservname** (funzione), 452  
**getsid** (funzione)  
    definizione di, 225  
**getsockname** (funzione), 386, 387  
    definizione di, 386  
**getsockopt** (funzione), 457, 458, 460, 461, 463  
    definizione di, 458  
**gettimeofday** (funzione), 178  
    definizione di, 178  
**getty** (funzione), 230  
**getuid** (funzione)  
    definizione di, 55  
**getutent** (funzione), 170  
    definizione di, 169  
**getutent\_r** (funzione), 171  
**getutid** (funzione), 170  
    definizione di, 169  
**getutid\_r** (funzione), 171  
**getutline** (funzione), 170  
    definizione di, 169  
**getutline\_r** (funzione), 171  
**getutxent** (funzione), 171  
**getutxid** (funzione), 171  
**getutxline** (funzione), 171  
**GETVAL** (costante), 310, 314  
**getw** (funzione)  
    definizione di, 141  
**getwc** (funzione)  
    definizione di, 141  
**getwchar** (funzione)  
    definizione di, 141  
**GETZCNT** (costante), 310  
**gid\_t** (tipo), 8  
**gmtime** (funzione), 181  
**group** (struttura dati), 168  
    definizione di, 168  
**herror** (funzione), 440  
    definizione di, 438  
**HOST\_NOT\_FOUND** (costante), 438  
**hostent** (struttura dati), 438, 439, 441–444  
    definizione di, 439  
**hsearch** (funzione), 11  
**hstrerror** (funzione)  
    definizione di, 438  
**htonl** (funzione), 381  
    definizione di, 368  
**hton** (funzione), 389  
    definizione di, 368  
**HUPCL** (costante), 240  
**HZ** (costante), 39, 176, 207  
**ICANON** (costante), 240, 241  
**ICRNL** (costante), 238  
**IEXTEN** (costante), 241, 242  
**IFNAMSIZ** (costante), 461  
**IFTODT** (macro), 91  
**IGNBRK** (costante), 238  
**IGNCR** (costante), 238  
**IGNPAR** (costante), 238  
**IMAXBEL** (costante), 238  
**in6\_addr** (struttura dati), 370, 442  
**in6addr\_any** (costante), 381  
**IN6ADDR\_ANY\_INIT** (costante), 448  
**in6addr\_loopback** (costante), 381  
**IN6ADDR\_ANY\_INIT** (costante), 381  
**IN6ADDR\_LOOPBACK\_INIT** (costante), 381  
**in\_addr** (struttura dati), 369, 370, 442  
**in\_addr\_t** (tipo), 362, 369  
**in\_port\_t** (tipo), 362  
**INADDR\_ANY** (costante), 379, 381, 448, 472  
**INADDR\_BROADCAST** (costante), 381  
**INADDR\_LOOPBACK** (costante), 381  
**INADDR\_NONE** (costante), 369, 381  
**INET6\_ADDRSTRLEN** (costante), 369  
**inet\_addr** (funzione), 368, 369  
    definizione di, 368  
**INET\_ADDRSTRLEN** (costante), 369  
**INET\_ANY** (costante), 392  
**inet\_aton** (funzione), 368, 369

**definizione di**, 368  
**inet\_ntoa** (funzione), 368, 369  
    **definizione di**, 368  
**inet\_ntop** (funzione), 369, 370, 394, 441, 451  
    **definizione di**, 369  
**inet\_pton** (funzione), 369, 389, 442, 454, 470  
    **definizione di**, 369  
**INIT\_PROCESS** (costante), 170  
**initgroups** (funzione), 61, 230  
    **definizione di**, 61  
**INLCR** (costante), 238  
**inline** (direttiva), 214  
**ino\_t** (tipo), 8  
**inode**, 8, 73, 75–81, 83, 84, 88, 89, 95, 98, 101–104, 115–117, 124, 127, 269, 272, 286, 294, 306  
**inode** (struttura dati), 269  
**INPCK** (costante), 238  
**int** (tipo), 467, 468  
**int16\_t** (tipo), 362  
**int32\_t** (tipo), 362  
**int8\_t** (tipo), 362  
**INT\_MAX** (costante), 156  
**INT\_MIN** (costante), 156  
**intmax\_t** (tipo), 146  
**ioctl** (funzione), 131, 132, 200, 227, 236, 300, 332, 463  
    **definizione di**, 131  
**iov\_base** (funzione), 261  
**iovec** (struttura dati), 261  
    **definizione di**, 261  
**IPC\_CREAT** (costante), 297, 298  
**IPC\_CREATE** (costante), 314, 322  
**IPC\_EXCL** (costante), 297, 298  
**IPC\_NOWAIT** (costante), 300–302, 311, 312  
**ipc\_perm** (struttura dati), 293–295  
    **definizione di**, 293  
**IPC\_PRIVATE** (costante), 297, 298  
**IPC\_RMID** (costante), 300, 309, 315, 318  
**IPC\_SET** (costante), 300, 309, 310, 318  
**IPC\_STAT** (costante), 300, 309, 318  
**IPCMNI** (costante), 296  
**isatty** (funzione)  
    **definizione di**, 236  
**ISIG** (costante), 241, 242  
**ISTRIP** (costante), 238  
**ITIMER\_PROF** (costante), 204  
**ITIMER\_REAL** (costante), 204  
**ITIMER\_VIRT** (costante), 205  
**ITIMER\_VIRTUAL** (costante), 204  
**itimerval** (struttura dati), 204  
    **definizione di**, 204  
**IUCLC** (costante), 238  
**IXANY** (costante), 238  
**IXOFF** (costante), 238  
**IXON** (costante), 238, 242  
**jmp\_buf** (tipo), 34, 219  
**key\_t** (tipo), 8, 293  
**kill** (funzione), 189, 191, 192, 198, 201–203, 213, 214, 221  
    **definizione di**, 202  
**killpg** (funzione)  
    **definizione di**, 202  
**L\_ctermid** (costante), 237  
**L\_INCR** (costante), 121  
**L\_SET** (costante), 121  
**L\_tmpnam** (costante), 97  
**L\_XTND** (costante), 121  
**lchown** (funzione), 85, 102, 111  
    **definizione di**, 111  
**LENGTH** (costante), 301, 302  
**link** (funzione), 76, 78, 81, 82, 84, 85, 102, 119, 328  
**LINK\_MAX** (costante), 82, 159, 504  
**linked list**, 269  
**lio\_listio** (funzione), 257  
    **definizione di**, 260  
**LIO\_NOP** (costante), 260  
**LIO\_NOWAIT** (costante), 260  
**LIO\_READ** (costante), 260  
**LIO\_WAIT** (costante), 260  
**LIO\_WRITE** (costante), 260  
**listen** (funzione), 371, 381, 383, 386, 392, 401, 404, 405, 463, 465, 472  
    **definizione di**, 383  
**LLONG\_MAX** (costante), 156  
**LLONG\_MIN** (costante), 156  
**localtime** (funzione), 181, 182  
**LOCK\_EX** (costante), 268, 277  
**LOCK\_NB** (costante), 268, 276, 277  
**LOCK\_SH** (costante), 268, 277  
**LOCK\_UN** (costante), 268, 277  
**lockf** (funzione), 276, 277  
    **definizione di**, 276  
**LockFile** (funzione), 328  
**LockMutex** (funzione), 329, 331

**loff\_t** (tipo), 8  
**LOG\_ALERT** (costante), 234  
**LOG\_AUTH** (costante), 233  
**LOG\_AUTHPRIV** (costante), 233  
**LOG\_CONS** (costante), 233  
**LOG\_CRIT** (costante), 234  
**LOG\_CRON** (costante), 233  
**LOG\_DAEMON** (costante), 233  
**LOG\_DEBUG** (costante), 234  
**LOG\_EMERG** (costante), 234  
**LOG\_ERR** (costante), 234  
**LOG\_FTP** (costante), 233  
**LOG\_INFO** (costante), 234  
**LOG\_KERN** (costante), 233  
**LOG\_LOCAL0** (costante), 233  
**LOG\_LOCAL7** (costante), 233  
**LOG\_LPR** (costante), 233  
**LOG\_MAIL** (costante), 233  
**LOG\_MASK(p)** (macro), 234  
**LOG\_NDELAY** (costante), 233  
**LOG\_NEWS** (costante), 233  
**LOG\_NOTICE** (costante), 234  
**LOG\_NOWAIT** (costante), 233  
**LOG\_ODELAY** (costante), 233  
**LOG\_PERROR** (costante), 233  
**LOG\_PID** (costante), 233  
**LOG\_SYSLOG** (costante), 233  
**LOG\_UPTO(p)** (macro), 234  
**LOG\_USER** (costante), 233  
**LOG\_UUCP** (costante), 233  
**LOG\_WARNING** (costante), 234  
**LOGIN\_PROCESS** (costante), 170  
**logwtmp** (funzione)  
    definizione di, 171  
**LONG\_MAX** (costante), 156  
**LONG\_MIN** (costante), 156  
**longjmp** (funzione), 22, 34, 35, 209, 210,  
    216, 219  
    definizione di, 34  
**lseek** (funzione), 101, 117, 120–124, 126,  
    127, 148, 256, 271, 284  
    definizione di, 120  
**lstat** (funzione), 85, 99  
    definizione di, 99  
**main** (funzione), 13–16, 25, 26, 46, 47, 50, 52  
**malloc** (funzione), 20–22, 33, 92, 136, 144,  
    150, 174  
    definizione di, 20  
**MAP\_ANON** (costante), 264  
**MAP\_ANONYMOUS** (costante), 263, 264, 331  
**MAP\_DENYWRITE** (costante), 263, 264  
**MAP\_EXECUTABLE** (costante), 264  
**MAP\_FAILED** (costante), 263  
**MAP\_FILE** (costante), 264  
**MAP\_FIXED** (costante), 264  
**MAP\_GROWSDOWN** (costante), 264  
**MAP\_LOCKED** (costante), 264  
**MAP\_NORESERVE** (costante), 264  
**MAP\_PRIVATE** (costante), 263–265  
**MAP\_SHARED** (costante), 263–265, 278, 331  
**MAX\_CANON** (costante), 159, 235  
**MAX\_INPUT** (costante), 159, 235  
**MAX\_IOVEC** (costante), 261  
**MAXLINE** (costante), 397  
**MAXSYMLINKS** (costante), 87  
**MB\_LEN\_MAX** (costante), 156  
**MCL\_CURRENT** (costante), 25  
**MCL\_FUTURE** (costante), 25  
memoria virtuale, 4, 17, 23–25, 172, 173,  
    262, 263, 265, 318, 338  
**memset** (funzione), 322, 324, 340, 470  
**MINSIGSTKSZ** (costante), 218  
**mkdir** (funzione), 85, 98, 102  
    definizione di, 87  
**mkdtemp** (funzione)  
    definizione di, 98  
**mkfifo** (funzione), 85, 102, 286, 289  
    definizione di, 89  
**mknod** (funzione), 85, 89, 286  
    definizione di, 88  
**mkstemp** (funzione), 97  
    definizione di, 98  
**mktemp** (funzione), 98  
    definizione di, 97  
**mktimes** (funzione), 181, 182  
**mllock** (funzione), 24  
    definizione di, 24  
**mlockall** (funzione), 25  
    definizione di, 24  
**mmap** (funzione), 174, 263–265, 278, 339, 340  
    definizione di, 263  
**MNT\_FORCE** (costante), 165  
**mntent** (struttura dati), 166  
**mode\_t** (tipo), 8, 109  
**mount** (funzione), 164, 277  
    definizione di, 163  
**mq\_attr** (struttura dati), 334, 335  
    definizione di, 334  
**mq\_close** (funzione)  
    definizione di, 334

**mq\_getaddr** (funzione), 336  
**mq\_getattr** (funzione), 335  
  definizione di, 335  
**MQ\_MAXMSG** (costante), 334  
**MQ\_MSGSIZE** (costante), 334  
**mq\_notify** (funzione), 337  
  definizione di, 337  
**mq\_open** (funzione), 333, 334  
  definizione di, 333  
**MQ\_PRIO\_MAX** (costante), 336  
**mq\_receive** (funzione), 333, 337  
  definizione di, 336  
**mq\_send** (funzione), 333, 336  
  definizione di, 335  
**mq\_setattr** (funzione), 335  
  definizione di, 335  
**mq\_timedreceive** (funzione)  
  definizione di, 336  
**mq\_timedsend** (funzione)  
  definizione di, 335  
**mq\_unlink** (funzione), 335  
  definizione di, 334  
**mqd\_t** (tipo), 333  
**MS\_ASYNC** (costante), 266  
**MS\_BIND** (costante), 164  
**MS\_INVALIDATE** (costante), 266  
**MS\_MANDLOCK** (costante), 164  
**MS\_MGC\_MSK** (costante), 164  
**MS\_MGC\_VAL** (costante), 164  
**MS\_MOVE** (costante), 164  
**MS\_NOATIME** (costante), 164  
**MS\_NODEV** (costante), 163, 164  
**MS\_NODIRATIME** (costante), 164  
**MS\_NOEXEC** (costante), 164  
**MS\_NOSUID** (costante), 164  
**MS\_RDONLY** (costante), 164  
**MS\_REMOUNT** (costante), 164  
**MS\_SYNC** (costante), 266  
**MS\_SYNCHRONOUS** (costante), 164  
**msg** (struttura dati), 301  
**MSG\_EXCEPT** (costante), 302  
**MSG\_NOERROR** (costante), 302, 303  
**MSG\_NOSIGNAL** (costante), 467, 468  
**MSG\_OOB** (costante), 460  
**MSG\_R** (costante), 294  
**MSG\_W** (costante), 294  
**msgbuf** (struttura dati), 301  
  definizione di, 301  
**msgctl** (funzione)  
  definizione di, 300  
**msgget** (funzione), 299, 303, 305, 307, 316  
  definizione di, 297  
**msgid\_ds** (struttura dati), 299  
**MSGMAX** (costante), 298, 300, 301  
**MSGMNB** (costante), 298–300  
**MSGMNI** (costante), 296–298  
**msgrcv** (funzione), 303, 305  
  definizione di, 302  
**msgsnd** (funzione), 301  
  definizione di, 300  
**msqid\_ds** (struttura dati), 300–302  
  definizione di, 299  
**msync** (funzione), 264–266  
  definizione di, 266  
**munlock** (funzione), 24  
  definizione di, 24  
**munlockall** (funzione)  
  definizione di, 24  
**munmap** (funzione), 340  
  definizione di, 266  
**MutexCreate** (funzione), 314  
**MutexFind** (funzione), 314, 326  
**MutexLock** (funzione), 314–316, 324–326  
**MutexRead** (funzione), 314, 316  
**MutexRemove** (funzione), 314, 325  
**MutexUnlock** (funzione), 314–316, 325, 326  
**NAME\_MAX** (costante), 91, 159  
**nanosleep** (funzione), 207  
  definizione di, 207  
**NCCS** (costante), 241  
**NET\_TCP\_MAX\_SYN\_BACKLOG** (costante), 384  
**netent** (struttura dati), 444  
**NEW\_TIME** (costante), 170  
**NGROUP\_MAX** (costante), 158  
**NGROUPS\_MAX** (costante), 59, 157  
**NI\_DGRAM** (costante), 452  
**NI\_MAXHOST** (costante), 452  
**NI\_MAXSERV** (costante), 452  
**NI\_NAMEREQD** (costante), 452  
**NI\_NOFQDN** (costante), 452  
**NI\_NUMERICHOST** (costante), 452  
**NI\_NUMERICSERV** (costante), 452  
**nice** (funzione)  
  definizione di, 63  
**NLDLY** (costante), 239  
**nlink\_t** (tipo), 8  
**NO\_ADDRESS** (costante), 438  
**NO\_DATA** (costante), 438  
**NO\_RECOVERY** (costante), 438  
**NOFLSH** (costante), 241

**NSIG** (costante), 193  
**ntohl** (funzione)  
    definizione di, 368  
**ntohs** (funzione)  
    definizione di, 368  
**ntp\_adjtime** (funzione), 180  
**NULL** (costante), 442–445, 447–449, 452, 456, 458, 468, 476  
  
**O\_ACCMODE** (costante), 131  
**O\_APPEND** (costante), 118, 120, 123, 124, 126, 129  
**O\_ASYNC** (costante), 118, 129, 255  
**O\_CREAT** (costante), 117, 118, 126, 328, 333, 334, 339, 340  
**O\_CREATE** (costante), 102  
**O\_DIRECT** (costante), 118  
**O\_DIRECTORY** (costante), 117–119  
**O\_DSYNC** (costante), 118, 259  
**O\_EXCL** (costante), 97, 98, 117, 118, 126, 137, 328, 333, 339  
**O\_EXLOCK** (costante), 118  
**O\_FSYNC** (costante), 118  
**O\_LARGEFILE** (costante), 118  
**O\_NDELAY** (costante), 118  
**O\_NOATIME** (costante), 118  
**O\_NOBLOCK** (costante), 117, 240  
**O\_NOCTTY** (costante), 118, 227, 231  
**O\_NOFOLLOW** (costante), 117–119  
**O\_NONBLOCK** (costante), 118, 121, 123, 129, 249, 277, 278, 333, 335, 336, 386  
**O\_RDONLY** (costante), 118, 131, 333, 339  
**O\_RDWR** (costante), 118, 131, 289, 333, 339  
**O\_READ** (costante), 118  
**O\_RSYNC** (costante), 118  
**O\_SHLOCK** (costante), 118  
**O\_SYNC** (costante), 118, 259  
**O\_TRUNC** (costante), 101, 102, 118, 278, 339  
**O\_WRITE** (costante), 118  
**O\_WRONLY** (costante), 117, 118, 131, 333  
**OCRNL** (costante), 239  
**OFDEL** (costante), 239  
**off\_t** (tipo), 8, 148, 149  
**offset** (funzione), 122  
**OFILL** (costante), 239  
**OLCUC** (costante), 239  
**OLD\_TIME** (costante), 170  
**on\_exit** (funzione), 15, 205  
    definizione di, 15  
**ONLCR** (costante), 239  
**ONLRET** (costante), 239  
  
**ONOCR** (costante), 239  
**open** (funzione), 76, 85, 86, 98, 102, 115, 117–119, 125–127, 129, 130, 136, 137, 240, 249, 273, 278, 286, 289, 328, 329, 333, 334, 339  
    definizione di, 117  
**OPEN\_MAX** (costante), 157  
**opendir** (funzione), 53, 85, 118  
    definizione di, 89  
**openlog** (funzione), 233, 234  
    definizione di, 232  
**openpty** (funzione), 248  
**OPOST** (costante), 239  
  
**P\_tmpdir** (costante), 97  
**PAGE\_SIZE** (costante), 174, 316, 317, 320  
**PAGECACHE\_SIZE** (costante), 339  
**PAGESIZE** (costante), 24, 266  
paginazione, 17, 23, 24, 174, 262, 264  
**PARENB** (costante), 240  
**PARMRK** (costante), 238  
**PARODD** (costante), 240  
**passwd** (struttura dati), 167  
    definizione di, 167  
**PATH\_MAX** (costante), 96, 159, 332  
**pathconf** (funzione), 85, 160  
    definizione di, 160  
**pathname** (funzione), 88  
**pause** (funzione), 200, 206, 209, 210, 216  
    definizione di, 206  
**pclose** (funzione), 283, 285, 286  
    definizione di, 284  
**PENDIN** (costante), 241  
**perror** (funzione), 184, 186, 199, 389, 400, 438, 503  
    definizione di, 185  
**PF\_APPLETALK** (costante), 359, 361, 364  
**PF\_ASH** (costante), 359  
**PF\_ATMPVC** (costante), 359, 361  
**PF\_ATMSVC** (costante), 359  
**PF\_AX25** (costante), 359, 361  
**PF\_BLUETOOTH** (costante), 359  
**PF\_BRIDGE** (costante), 359  
**PF\_DECnet** (costante), 359  
**PF\_ECONET** (costante), 359  
**PF\_FILE** (costante), 359  
**PF\_INET** (costante), 359, 361, 362, 447, 465  
**PF\_INET6** (costante), 359, 361, 363, 447, 465  
**PF\_INTERP** (costante), 51  
**PF\_IPX** (costante), 359, 361  
**PF\_IRDA** (costante), 359

**PF\_KEY** (costante), 359  
**PF\_LOCAL** (costante), 359, 363  
**PF\_MAX** (costante), 360  
**PF\_NETBEUI** (costante), 359  
**PF\_NETLINK** (costante), 359, 361  
**PF\_NETROM** (costante), 359  
**PF\_PACKET** (costante), 359, 361, 365  
**PF\_PPPOX** (costante), 359  
**PF\_ROSE** (costante), 359  
**PF\_SECURITY** (costante), 359  
**PF\_SNA** (costante), 359  
**PF\_UNIX** (costante), 359, 361, 363  
**PF\_UNSPEC** (costante), 359, 447  
**PF\_WANPIPE** (costante), 359  
**PF\_X25** (costante), 359, 361  
**PID\_MAX** (costante), 40  
**pid\_t** (tipo), 8, 40, 64, 224  
**pipe** (funzione), 102, 136, 284, 292  
     definizione di, 279  
**PIPE\_BUF** (costante), 159, 280, 281, 286, 290, 388  
**poll** (funzione), 253–255, 303, 426–428  
     definizione di, 253  
**POLLERR** (costante), 254, 427  
**pollfd** (struttura dati), 253, 254, 427  
     definizione di, 254  
**POLLHUP** (costante), 254, 427  
**POLLIN** (costante), 254, 426  
**POLLMSG** (costante), 254  
**POLLNVAL** (costante), 254  
**POLLOUT** (costante), 254, 427  
**POLLPRI** (costante), 254, 426  
**POLLRDBAND** (costante), 254, 426  
**POLLRDBAND** (macro), 254  
**POLLRDNORM** (costante), 254, 426, 427  
**POLLRDNORM** (macro), 254  
**POLLWRBAND** (costante), 254  
**POLLWRNORM** (costante), 254  
**popen** (funzione), 283–285  
     definizione di, 284  
**POSIXLY\_CORRECT** (macro), 27  
**pread** (funzione), 122, 123  
     definizione di, 122  
**PrintErr** (funzione), 400  
**printf** (funzione), 17, 33, 44, 145–148, 186, 232, 234  
     definizione di, 145  
**printk** (funzione), 232  
**PRI\_O\_MAX** (costante), 63  
**PRI\_O\_MIN** (costante), 63  
**PRI\_O\_PRGR** (costante), 64  
**PRI\_O\_PROCESS** (costante), 64  
**PRI\_O\_USER** (costante), 64  
**PROT\_EXEC** (costante), 263  
**PROT\_NONE** (costante), 263  
**PROT\_READ** (costante), 263  
**PROT\_WRITE** (costante), 263, 265  
**protoent** (struttura dati), 444  
**pselect** (funzione), 250, 252, 253  
     definizione di, 252  
**psignal** (funzione), 198, 199, 252  
     definizione di, 199  
**PT\_INTERP** (costante), 53  
**ptrdiff\_t** (tipo), 8, 146  
**putc** (funzione), 153  
     definizione di, 141  
**putchar** (funzione)  
     definizione di, 141  
**putenv** (funzione), 29, 30  
     definizione di, 29  
**putgrent** (funzione), 168  
**putpwent** (funzione), 168  
**puts** (funzione), 143  
     definizione di, 143  
**pututline** (funzione), 171  
     definizione di, 169  
**pututxline** (funzione), 171  
**putw** (funzione)  
     definizione di, 141  
**pwrite** (funzione), 122  
     definizione di, 123  
**qsort** (funzione), 92  
**R\_OK** (costante), 109  
**raise** (funzione), 189, 191, 201, 202, 205  
     definizione di, 202  
**read** (funzione), 102, 117–119, 121–123, 133, 134, 235, 242, 247, 248, 251, 258, 267, 277, 280, 289, 360, 386–388, 390, 397, 400–402, 406, 410, 412, 414, 415, 418, 425, 426, 460, 461, 466, 468, 476  
     definizione di, 121  
**readdir** (funzione), 90  
     definizione di, 90  
**readdir\_r** (funzione), 90  
**readlink** (funzione), 85  
     definizione di, 86  
**ReadMutex** (funzione), 329  
**readv** (funzione), 461

definizione di, 261  
**realloc** (funzione), 20, 21, 144  
    definizione di, 20  
**recv** (funzione), 366, 461  
**recvfrom** (funzione), 366, 404, 461, 465, 466,  
    468, 470, 472, 474–476  
    definizione di, 468  
**recvmsg** (funzione), 366, 460, 461  
**register** (direttiva), 31, 35  
**remove** (funzione), 83, 85, 102  
    definizione di, 83  
**RemoveMutex** (funzione), 329  
**RemoveShm** (funzione), 340  
**rename** (funzione), 79, 83–85, 102  
    definizione di, 84  
**RES\_AAONLY** (costante), 435  
**RES\_BLAST** (costante), 435  
**RES\_DEBUG** (costante), 435  
**RES\_DEFAULT** (costante), 434, 435  
**RES\_DEFNAMES** (costante), 435, 436  
**RES\_DNSRCH** (costante), 435, 436  
**RES\_IGNTC** (costante), 435  
**RES\_INIT** (costante), 435  
**res\_init** (funzione), 434, 435, 439  
    definizione di, 434  
**RES\_INSECURE1** (costante), 435  
**RES\_INSECURE2** (costante), 435  
**RES\_KEEPSIG** (costante), 435  
**RES\_NOALIASES** (costante), 435  
**RES\_NOCHECKNAME** (costante), 435  
**RES\_PRIMARY** (costante), 435  
**res\_query** (funzione), 436, 437  
    definizione di, 435  
**RES\_RECURSE** (costante), 435  
**RES\_ROTATE** (costante), 435  
**res\_search** (funzione), 435  
    definizione di, 436  
**RES\_STAYOPEN** (costante), 435  
**RES\_TIMEOUT** (costante), 435  
**RES\_USE\_INET6** (costante), 435, 439  
**RES\_USEVC** (costante), 435  
**rewind** (funzione), 137, 148  
    definizione di, 148  
**rewaddir** (funzione)  
    definizione di, 92  
**RLIM\_INFINITY** (costante), 173  
**rlim\_t** (tipo), 8  
**rlimit** (struttura dati), 173  
    definizione di, 173  
**RLIMIT\_AS** (costante), 174  
**RLIMIT\_CORE** (costante), 174  
**RLIMIT\_CPU** (costante), 174  
**RLIMIT\_DATA** (costante), 174  
**RLIMIT\_FSIZE** (costante), 174  
**RLIMIT\_MEMLOCK** (costante), 174  
**RLIMIT\_NOFILE** (costante), 174  
**RLIMIT\_NOFILE** (macro), 253  
**RLIMIT\_NPROC** (costante), 174  
**RLIMIT\_RSS** (costante), 174  
**RLIMIT\_STACK** (costante), 174  
**rmdir** (funzione), 83, 102  
    definizione di, 88  
**RUN\_LVL** (costante), 170  
**rusage** (struttura dati), 51, 171, 172, 176  
    definizione di, 172  
**RUSAGE\_CHILDREN** (costante), 172  
**RUSAGE\_SELF** (costante), 172  
  
**S\_APPEND** (costante), 164  
**S\_IFBLK** (costante), 89, 100  
**S\_IFCHR** (costante), 89, 100  
**S\_IFDIR** (costante), 100  
**S\_IFIFO** (costante), 89, 100  
**S\_IFLNK** (costante), 100  
**S\_IFMT** (costante), 100  
**S\_IFREG** (costante), 89, 100  
**S\_IFSOCK** (costante), 100  
**S\_IGID** (costante), 107  
**S\_IMMUTABLE** (costante), 164  
**S\_IRGRP** (costante), 100, 105, 110  
**S\_IROTH** (costante), 100, 105, 110  
**S\_IRUSR** (costante), 100, 105, 110  
**S\_IWXG** (costante), 110  
**S\_IRWXO** (costante), 110  
**S\_IRWXU** (costante), 110  
**S\_ISBLK(m)** (macro), 100  
**S\_ISCHR(m)** (macro), 100  
**S\_ISDIR(m)** (macro), 100  
**S\_ISFIFO(m)** (macro), 100  
**S\_ISGID** (costante), 100, 106, 110  
**S\_ISLNK(m)** (macro), 100  
**S\_ISREG(m)** (macro), 100  
**S\_ISSOCK(m)** (macro), 100  
**S\_ISUID** (costante), 100, 106, 107, 110  
**S\_ISVTX** (costante), 100, 107, 110  
**S\_IWGRP** (costante), 100, 105, 110  
**S\_IWOTH** (costante), 100, 105, 110  
**S\_IWUSR** (costante), 100, 105, 110  
**S\_IXGRP** (costante), 100, 105, 110  
**S\_IXOTH** (costante), 100, 105, 110  
**S\_IXUSR** (costante), 100, 105, 110

**S\_WRITE** (costante), 164  
**sa\_family\_t** (tipo), 362  
**SA\_NOCLDSTOP** (costante), 213  
**SA\_NODEFER** (costante), 213  
**SA\_NOMASK** (costante), 213  
**SA\_ONESHOT** (costante), 213  
**SA\_ONSTACK** (costante), 213, 218  
**SA\_RESETHAND** (costante), 213  
**SA\_RESTART** (costante), 213, 403, 404  
**SA\_SIGINFO** (costante), 130, 213, 220, 221, 257  
salto non-locale, 33–35, 219  
**sbrk** (funzione), 22  
    definizione di, 23  
**scandir** (funzione)  
    definizione di, 92  
**scanf** (funzione), 147  
    definizione di, 147  
**SCHAR\_MAX** (costante), 156  
**SCHAR\_MIN** (costante), 156  
**SCHED\_FIFO** (costante), 65, 66, 207  
**sched\_get\_priority\_max** (funzione)  
    definizione di, 65  
**sched\_get\_priority\_min** (funzione)  
    definizione di, 65  
**sched\_getparam** (funzione)  
    definizione di, 66  
**sched\_getscheduler** (funzione)  
    definizione di, 66  
**SCHED\_OTHER** (costante), 65, 66  
**sched\_param** (struttura dati), 65  
    definizione di, 66  
**SCHED\_RR** (costante), 65, 207  
**sched\_rr\_get\_interval** (funzione)  
    definizione di, 67  
**sched\_setparam** (funzione), 67  
    definizione di, 66  
**sched\_setscheduler** (funzione), 65, 67  
    definizione di, 65  
**sched\_yield** (funzione)  
    definizione di, 67  
**SCM\_CREDENTIALS** (costante), 461  
**SEEK\_CUR** (costante), 121, 271  
**SEEK\_END** (costante), 121, 271  
**SEEK\_SET** (costante), 121, 271  
**seekdir** (funzione), 91  
**select** (funzione), 92, 250–253, 255, 303, 404, 415–419, 421, 423–427, 429, 460, 461, 507  
    definizione di, 250  
**sem** (struttura dati), 308, 310, 311, 313  
    definizione di, 308  
**sem\_queue** (struttura dati), 313  
**SEM\_UNDO** (costante), 311, 312, 314  
**sem\_undo** (struttura dati), 313, 314  
**SEMAEM** (costante), 308  
**sembuf** (struttura dati), 311, 313  
    definizione di, 311  
**semctl** (funzione), 300, 307, 309–311, 314, 315  
    definizione di, 309  
**semget** (funzione), 307, 308, 314, 316  
    definizione di, 307  
**semid\_ds** (struttura dati), 308–310, 313  
    definizione di, 307  
**SEMMNI** (costante), 296, 307, 308  
**SEMMNS** (costante), 307, 308  
**SEMMNU** (costante), 308  
**SEMMSL** (costante), 307, 308  
**semop** (funzione), 308, 311, 312, 314, 315  
    definizione di, 311  
**SEMOPM** (costante), 308, 311  
**SEMUME** (costante), 308  
**semun** (struttura dati), 309  
    definizione di, 309  
**semunion** (struttura dati), 314  
**SEMVMX** (costante), 308, 309, 311, 312  
**send** (funzione), 461  
**sendfrom** (funzione), 461  
**sendmsg** (funzione), 461  
**sendto** (funzione), 448, 465, 466, 468, 470, 472, 473, 475, 476  
    definizione di, 467  
**servent** (struttura dati), 444, 445  
    definizione di, 445  
**SETALL** (costante), 309, 310  
**setbuf** (funzione), 150  
    definizione di, 151  
**setbuffer** (funzione)  
    definizione di, 151  
**setegid** (funzione), 58  
    definizione di, 58  
**setenv** (funzione), 29, 30  
    definizione di, 29  
**seteuid** (funzione), 58  
    definizione di, 58  
**setfsgid** (funzione), 59  
    definizione di, 59  
**setfsuid** (funzione), 59  
    definizione di, 59

**setgid** (funzione), 56, 57, 230, 398  
    definizione di, 56  
**setgrent** (funzione), 168  
**setgroups** (funzione), 59, 61  
    definizione di, 60  
**sethostent** (funzione), 446  
    definizione di, 442  
**setitimer** (funzione), 204, 205  
    definizione di, 204  
**setjmp** (funzione), 34, 35, 210, 219  
    definizione di, 34  
**setlinebuf** (funzione)  
    definizione di, 151  
**setlogmask** (funzione)  
    definizione di, 234  
**setnetent** (funzione), 446  
**setpgid** (funzione), 225, 229  
    definizione di, 225  
**setpgrp** (funzione), 226  
    definizione di, 225  
**setpriority** (funzione), 64  
    definizione di, 64  
**setprotoent** (funzione), 446  
**setpwent** (funzione), 168  
**setregid** (funzione), 57  
    definizione di, 57  
**setresgid** (funzione), 58  
    definizione di, 58  
**setresuid** (funzione), 58  
    definizione di, 58  
**setreuid** (funzione), 57  
    definizione di, 57  
**setrlimit** (funzione), 174  
    definizione di, 173  
**setservent** (funzione), 445, 446  
    definizione di, 445  
**setsid** (funzione), 226, 228, 230, 231  
    definizione di, 226  
**setsockopt** (funzione), 457, 458, 460, 463  
    definizione di, 457  
**SetTermAttr** (funzione), 243, 244  
**settimeofday** (funzione), 178  
    definizione di, 178  
**setuid** (funzione), 56–58, 230, 398  
    definizione di, 56  
**setutent** (funzione)  
    definizione di, 169  
**setutxent** (funzione), 171  
**SETVAL** (costante), 309, 310, 314  
**setvbuf** (funzione), 151  
    definizione di, 150  
sezioni critiche, 68, 306  
**SHM\_LOCK** (costante), 318  
**shm\_open** (funzione), 339, 340  
    definizione di, 339  
**SHM\_RDONLY** (costante), 320  
**SHM\_RND** (costante), 320  
**shm\_unlink** (funzione), 340  
    definizione di, 339  
**SHM\_UNLOCK** (costante), 318  
**shmaddr** (funzione), 320  
**SHMALL** (costante), 316, 317  
**shmat** (funzione), 319, 320, 322  
    definizione di, 318  
**ShmCreate** (funzione), 322, 324  
**shmctl** (funzione), 300  
    definizione di, 318  
**shmdt** (funzione), 318, 322  
    definizione di, 320  
**ShmFind** (funzione), 322, 326  
**shmget** (funzione), 316, 322  
    definizione di, 316  
**shmid\_ds** (struttura dati), 316, 318, 320  
    definizione di, 317  
**SHMLBA** (costante), 317, 320  
**SHMMAX** (costante), 316, 317  
**SHMMIN** (costante), 316, 317  
**SHMMNI** (costante), 296, 316, 317  
**ShmRemove** (funzione), 322, 325  
**SHMSEG** (costante), 317  
**SHRT\_MAX** (costante), 156  
**SHRT\_MIN** (costante), 156  
**SHUT\_RD** (macro), 420  
**SHUT\_RDWR** (macro), 420  
**SHUT\_WR** (macro), 420  
**shutdown** (funzione), 374, 388, 419–422, 427, 468  
    definizione di, 419, 420  
**SI\_MESGQ** (costante), 338  
**SI\_QUEUE** (costante), 221  
**SI\_SIGIO** (costante), 255  
**sig\_atomic\_t** (tipo), 68, 215  
**SIG\_BLOCK** (costante), 216  
**SIG\_DFL** (costante), 53, 199, 201  
**SIG\_ERR** (costante), 200  
**SIG\_IGN** (costante), 53, 199, 201, 207, 215  
**SIG\_SETMASK** (costante), 216  
**SIG\_UNBLOCK** (costante), 216  
**SIGABRT** (costante), 46, 194, 195, 203, 205

**sigaction** (funzione), 192, 200, 201, 212–216, 218, 220, 222, 403  
     definizione di, 212  
**sigaction** (struttura dati), 212–215, 403  
     definizione di, 213  
**sigaddset** (funzione)  
     definizione di, 211  
**SIGALRM** (costante), 194, 196, 203, 206, 207, 209, 213, 216, 217  
**sigaltstack** (funzione), 218  
     definizione di, 218  
**SIGBUS** (costante), 194, 195, 214, 265  
**SIGCHLD** (costante), 15, 46, 50, 53, 194, 197, 207, 208, 213, 214, 230, 403–407  
**SIGCLD** (costante), 194, 197, 207  
**SIGCONT** (costante), 46, 194, 197, 203, 228, 229  
**sigdelset** (funzione)  
     definizione di, 211  
**sigemptyset** (funzione), 212  
     definizione di, 211  
**SIGEMT** (costante), 194  
**SIGEV\_NONE** (costante), 257  
**SIGEV\_SIGNAL** (costante), 257, 337  
**SIGEV\_THREAD** (costante), 257, 337  
**sigevent** (struttura dati), 221, 257, 337  
     definizione di, 257  
**sigfillset** (funzione), 212  
     definizione di, 211  
**SIGFPE** (costante), 194, 201, 214  
**sighandler\_t** (tipo), 201  
**SIGHUP** (costante), 46, 194, 196, 228, 229, 240  
**SIGILL** (costante), 194, 201, 214  
**SIGINFO** (costante), 194, 198  
**siginfo\_t** (struttura dati), 130, 213, 220, 221, 255, 257, 338  
     definizione di, 214  
**SIGINT** (costante), 194, 195, 199, 228, 238, 241, 242  
**SIGIO** (costante), 118, 130, 194, 196, 198, 214, 255, 256  
**SIGIOT** (costante), 194  
**sigismember** (funzione), 211, 212  
     definizione di, 211  
**sigjmp\_buf** (tipo), 219  
**SIGKILL** (costante), 192, 194–196, 201, 212  
**siglongjmp** (funzione), 219  
     definizione di, 219  
**SIGLOST** (costante), 194, 198

**Signal** (funzione), 403, 404  
     definizione di, 215  
**signal** (funzione), 192, 200, 201, 212–215  
     definizione di, 200  
**SignalRestart** (funzione), 403  
     definizione di, 404  
**signo** (funzione), 221  
**sigpending** (funzione), 191  
     definizione di, 217  
**SIGPIPE** (costante), 123, 194, 197, 281, 291, 410, 416, 467, 468, 504  
**SIGPOLL** (costante), 194, 196, 214  
**sigprocmask** (funzione), 216, 217, 219, 252  
     definizione di, 215  
**SIGPROF** (costante), 194, 196, 204  
**SIGPWR** (costante), 194  
**sigqueue** (funzione), 222  
     definizione di, 221  
**SIGQUEUE\_MAX** (costante), 221  
**SIGQUIT** (costante), 194, 195, 199, 228, 241, 242  
**SIGRTMAX** (costante), 220  
**SIGRTMIN** (costante), 220  
**SIGSEGV** (costante), 18, 174, 194, 195, 201, 214, 219, 263–265, 320, 507  
**sigset\_t** (tipo), 8, 211, 212  
**sigsetjmp** (funzione), 219  
     definizione di, 219  
**SIGSTKFLT** (costante), 194  
**SIGSTKSZ** (costante), 218  
**SIGSTOP** (costante), 62, 192, 194, 197, 201, 212, 213  
**SIGSUSP** (costante), 241  
**sigsuspend** (funzione), 216, 217  
     definizione di, 216  
**SIGSYS** (costante), 194, 195  
**SIGTERM** (costante), 194–196, 228, 327, 409  
**sigtimedwait** (funzione), 222  
     definizione di, 222  
**SIGTRAP** (costante), 194, 195, 214  
**SIGTSTP** (costante), 194, 197, 213, 228, 242  
**SIGTTIN** (costante), 194, 197, 213, 227  
**SIGTTOU** (costante), 194, 197, 213, 227, 241, 243, 246  
**SIGUNUSED** (costante), 194  
**SIGURG** (costante), 130, 194, 196  
**SIGUSR1** (costante), 194, 198  
**SIGUSR2** (costante), 194, 198  
**sigval\_t** (struttura dati), 221, 337  
     definizione di, 221

SIGVTALRM (costante), 194, 196, 203  
**sigwait** (funzione), 222  
    definizione di, 222  
**sigwaitinfo** (funzione), 222  
    definizione di, 222  
**SIGWINCH** (costante), 194, 198  
**SIGXCPU** (costante), 174, 194, 198  
**SIGXFSZ** (costante), 174, 194, 198  
**size\_t** (tipo), 8, 146, 261, 447, 467  
**sleep** (funzione), 43, 206, 209, 210, 217, 325  
    definizione di, 206  
**snprintf** (funzione), 394  
    definizione di, 145  
**SO\_ACCEPTCONN** (costante), 459, 463  
**SO\_BINDTODEVICE** (costante), 459, 461  
**SO\_BROADCAST** (costante), 459, 463  
**SO\_BSDCOMPAT** (costante), 459, 461  
**SO\_DEBUG** (costante), 459, 461  
**SO\_DONTROUTE** (costante), 459, 463  
**SO\_ERROR** (costante), 415, 416, 459, 463  
**SO\_KEEPALIVE** (costante), 459  
**SO\_LINGER** (costante), 459, 463  
**SO\_OOBINLINE** (costante), 459, 460  
**SO\_PASSCRED** (costante), 459, 461  
**SO\_PEERCRED** (costante), 459, 461  
**SO\_PRIORITY** (costante), 459, 463  
**SO\_RCVBUF** (costante), 459, 463  
**SO\_RCVLOWAT** (costante), 415, 459–461  
**SO\_RCVTIMEO** (costante), 459–461  
**SO\_REUSEADDR** (costante), 459, 462  
**SO\_SNDBUF** (costante), 459, 463  
**SO\_SNDDLOWAT** (costante), 416, 459, 460  
**SO\_SNDFTIMEO** (costante), 459, 461  
**SO\_TYPE** (costante), 459, 463  
**SOCK\_DGRAM** (costante), 360, 361, 364, 365, 447, 465, 470, 473  
**SOCK\_PACKET** (costante), 360, 361  
**SOCK\_RAW** (costante), 360, 361, 364, 365  
**SOCK\_RDM** (costante), 360, 386  
**SOCK\_SEQPACKET** (costante), 360, 361, 383, 386  
**SOCK\_STREAM** (costante), 292, 360, 361, 383, 386, 389, 447, 463  
**sockaddr** (struttura dati), 361, 363, 447  
    definizione di, 361  
**sockaddr\_atalk** (struttura dati), 364  
    definizione di, 364  
**sockaddr\_in** (struttura dati), 362, 389, 392  
    definizione di, 362  
**sockaddr\_in6** (struttura dati), 363  
    definizione di, 363  
**sockaddr\_ll** (struttura dati), 365  
    definizione di, 365  
**sockaddr\_un** (struttura dati), 363  
    definizione di, 364  
**sockbind** (funzione), 455, 456  
**sockconn** (funzione), 453–456  
**socket**, 72–74, 83, 88, 100, 101, 120, 122, 136, 194, 196, 200, 214, 232, 249–251, 256, 291, 292, 327, 332, 350, 354, 355, 357–370  
**socket** (funzione), 358–361, 364, 365, 371, 380–383, 389, 401, 447, 454, 462, 465  
    definizione di, 358  
**socketpair** (funzione), 291, 292, 327, 363, 378  
    definizione di, 292  
**socklen\_t** (tipo), 362, 447  
**SOL\_ICMPV6** (costante), 458  
**SOL\_IP** (costante), 458  
**SOL\_IPV6** (costante), 458  
**SOL\_SOCKET** (costante), 457–459  
**SOL\_TCP** (costante), 458  
**SOMAXCONN** (costante), 384  
**sprintf** (funzione), 145, 147  
    definizione di, 145  
**SS\_DISABLE** (costante), 218, 219  
**SS\_ONSTACK** (costante), 218  
**sscanf** (funzione)  
    definizione di, 147  
**SSIZE\_MAX** (costante), 157, 158  
**ssize\_t** (tipo), 8, 146, 157, 158  
**stack\_t** (struttura dati), 218  
    definizione di, 219  
**stat** (funzione), 76, 78, 85, 94, 95, 98, 99, 101, 102, 104, 109, 126, 294, 325  
    definizione di, 99  
**stat** (struttura dati), 82, 85, 91, 99, 101, 104, 106, 133  
    definizione di, 99  
**statfs** (funzione), 165  
    definizione di, 165  
**statfs** (struttura dati), 165  
    definizione di, 165  
**STDERR\_FILENO** (costante), 117  
**STDIN\_FILENO** (costante), 117  
**STDOUT\_FILENO** (costante), 117  
**stime** (funzione), 178  
    definizione di, 178

**strcmp** (funzione), 93  
**strcoll** (funzione), 93  
**STREAM\_MAX** (costante), 157, 158  
**strerror** (funzione), 184, 185, 198, 438, 448, 503  
  definizione di, 184  
**strerror\_r** (funzione), 185  
**strftime** (funzione), 183  
  definizione di, 182  
**strsignal** (funzione), 198, 199  
  definizione di, 198  
**strtol** (funzione), 185  
**symlink** (funzione)  
  definizione di, 85  
**sync** (funzione), 120, 126, 137, 152  
  definizione di, 126  
**SYS\_NMLN** (costante), 161  
**sysconf** (funzione), 60, 156–159, 174–176  
  definizione di, 158  
**sysctl** (funzione), 161, 162, 221, 296, 298, 309, 317, 382, 384, 463  
  definizione di, 161  
**syslog** (funzione), 234, 400  
  definizione di, 233  
system call lente, 199, 213, 249, 256, 406  
**sysv\_signal** (funzione), 201

**T\_A** (costante), 437  
**T\_AAAA** (costante), 437  
**T\_AFSDB** (costante), 437  
**T\_ANY** (costante), 437  
**T\_ATMA** (costante), 437  
**T\_AXFR** (costante), 437  
**T\_CNAME** (costante), 437  
**T\_EID** (costante), 437  
**T\_GPOS** (costante), 437  
**T\_HINFO** (costante), 437  
**T\_ISDN** (costante), 437  
**T\_IXFR** (costante), 437  
**T\_KEY** (costante), 437  
**T\_LOC** (costante), 437  
**T\_MAILA** (costante), 437  
**T\_MAILB** (costante), 437  
**T\_MB** (costante), 437  
**T\_MD** (costante), 437  
**T\_MF** (costante), 437  
**T\_MG** (costante), 437  
**T\_MINFO** (costante), 437  
**T\_MR** (costante), 437  
**T\_MX** (costante), 437  
**T\_NAPTR** (costante), 437

**T\_NIMLOC** (costante), 437  
**T\_NS** (costante), 437  
**T\_NSAP** (costante), 437  
**T\_NSAP\_PTR** (costante), 437  
**T\_NULL** (costante), 437  
**T\_NXT** (costante), 437  
**T\_PTR** (costante), 437  
**T\_PX** (costante), 437  
**T\_RP** (costante), 437  
**T\_RT** (costante), 437  
**T\_SIG** (costante), 437  
**T\_SOA** (costante), 437  
**T\_SRV** (costante), 437  
**T\_TSIG** (costante), 437  
**T\_TXT** (costante), 437  
**T\_WKS** (costante), 437  
**T\_X25** (costante), 437  
**TABDLY** (costante), 239  
**task\_struct** (struttura dati), 38, 63, 115, 191, 192, 215, 224–226, 314  
**tcdrain** (funzione)  
  definizione di, 246  
**tcflag\_t** (tipo), 238  
**tcflow** (funzione), 247  
  definizione di, 246  
**tcflush** (funzione), 247  
  definizione di, 246  
**tcgetaddr** (funzione), 245  
**tcgetattr** (funzione), 243  
  definizione di, 242  
**tcgetpgrp** (funzione)  
  definizione di, 227  
**TCIFLUSH** (costante), 247  
**TCIOFF** (costante), 247  
**TCIOFLUSH** (costante), 247  
**TCION** (costante), 247  
**TCOFLUSH** (costante), 247  
**TCOOFF** (costante), 247  
**TCOON** (costante), 247  
**TCP\_MAXSEG** (costante), 372  
**TCP\_MSS** (costante), 356  
**TCSADRAIN** (costante), 243  
**TCSAFLUSH** (costante), 243  
**TCSANOW** (costante), 243  
**tcsendbreak** (funzione)  
  definizione di, 246  
**tcsetaddr** (funzione), 246  
**tcsetattr** (funzione), 243, 245  
  definizione di, 242  
**tcsetpgrp** (funzione)

definizione di, 227  
**telldir** (funzione), 91, 92  
**tmpfile** (funzione)  
    definizione di, 97  
**tmpnam** (funzione)  
    definizione di, 97  
**termios** (struttura dati), 237, 240, 242, 243, 245  
    definizione di, 237  
**time** (funzione), 178, 394  
    definizione di, 177  
**TIME\_BAD** (costante), 181  
**TIME\_DEL** (costante), 181  
**TIME\_INS** (costante), 181  
**TIME\_OK** (costante), 181  
**TIME\_OOP** (costante), 181  
**time\_t** (tipo), 8, 175, 177, 178, 181  
**TIME\_WAIT** (costante), 181  
**times** (funzione)  
    definizione di, 177  
**timespec** (struttura dati), 67, 178, 207, 252  
    definizione di, 178  
**timeval** (struttura dati), 172, 178, 204, 251, 460  
    definizione di, 178  
**timex** (struttura dati), 180  
    definizione di, 179  
**timezone** (struttura dati), 179  
**TIOCSCTTY** (costante), 227  
**tipo**  
    elementare, 8  
    opaco, 32, 34, 90, 134  
    primitivo, 8  
**tm** (struttura dati), 181  
    definizione di, 182  
**TMP\_MAX** (costante), 97  
**tmpfile** (funzione), 98  
**tmpnam** (funzione), 98  
    definizione di, 96, 97  
**tmpnam\_r** (funzione), 97  
**TMPNAME** (costante), 97  
**tmpname** (funzione), 40  
**tms** (struttura dati), 46, 177  
    definizione di, 177  
**TOSTOP** (costante), 241  
**truncate** (funzione), 85, 101, 102, 130, 278  
    definizione di, 101  
**TRY AGAIN** (costante), 438  
**ttynname** (funzione), 237  
    definizione di, 236  
**ttynname\_r** (funzione)  
    definizione di, 237  
**TZ** (costante), 182  
**TZNAME\_MAX** (costante), 157, 158  
**tzset** (funzione)  
    definizione di, 182  
**UCHAR\_MAX** (costante), 156  
**ucred** (struttura dati), 461  
**uid\_t** (tipo), 8, 64  
**uint16\_t** (tipo), 362  
**uint32\_t** (tipo), 362  
**uint8\_t** (tipo), 362  
**UINT\_MAX** (costante), 156  
**uintmax\_t** (tipo), 146  
**ULLONG\_MAX** (costante), 156  
**ULONG\_MAX** (costante), 156  
**umask** (funzione), 110  
    definizione di, 111  
**umount** (funzione), 165  
    definizione di, 164  
**umount2** (funzione)  
    definizione di, 165  
**uname** (funzione), 160–162  
    definizione di, 160  
**ungetc** (funzione), 142, 152  
    definizione di, 142  
**union** (direttiva), 309, 363  
**unlink** (funzione), 78, 81–85, 88, 102, 328, 329, 334, 340  
    definizione di, 82  
**UnlockFile** (funzione), 328  
**UnlockMutex** (funzione), 329, 331  
**unmap** (funzione), 264  
**unsetenv** (funzione), 29  
    definizione di, 29  
**UnSetTermAttr** (funzione), 243, 244  
**unsigned int** (tipo), 467  
**updwtmp** (funzione), 171  
    definizione di, 171  
**USER\_PROCESS** (costante), 170, 171  
**USHRT\_MAX** (costante), 156  
**usleep** (funzione)  
    definizione di, 206  
**utimbuf** (struttura dati)  
    definizione di, 103  
**utime** (funzione), 102, 103  
    definizione di, 103  
**utimebuf** (struttura dati), 103  
**utmp** (struttura dati), 170, 171  
    definizione di, 170

**utmpname** (funzione), 169  
     definizione di, 169  
**UTSLEN** (costante), 161  
**utsname** (struttura dati), 160  
     definizione di, 161

**va\_arg** (macro), 32, 33  
**va\_copy** (macro), 33  
**va\_end** (macro), 32, 33  
**va\_list** (macro), 32  
**va\_start** (macro), 32  
**variadic**, 31, 145  
**vasprintf** (funzione)  
     definizione di, 147  
**VDISCARD** (costante), 242  
**vdprintf** (funzione), 147  
**VEOF** (costante), 242  
**VEOL** (costante), 242  
**VEOL2** (costante), 242  
**VERASE** (costante), 242  
**versionsort** (funzione), 93  
     definizione di, 92  
**vfork** (funzione), 46  
**vfprintf** (funzione)  
     definizione di, 146  
**vfscanf** (funzione), 147  
**VINTR** (costante), 242  
**VKILL** (costante), 242  
**VLNEXT** (costante), 242  
**VMIN** (costante), 242, 247  
**volatile** (direttiva), 35, 68, 184  
**vprintf** (funzione)  
     definizione di, 146  
**VQUIT** (costante), 242  
**VREPRINT** (costante), 242  
**vscanf** (funzione), 147  
**vsnprintf** (funzione)  
     definizione di, 147  
**vsprintf** (funzione), 147  
     definizione di, 146  
**vsscanf** (funzione), 147  
**VSTART** (costante), 242  
**VSTOP** (costante), 242  
**VSUSP** (costante), 242  
**VSWTC** (costante), 242  
**VTDLY** (costante), 239  
**VTIME** (costante), 242, 247  
**VWERASE** (costante), 242

**W\_OK** (costante), 109

**wait** (funzione), 15, 39, 47–50, 54, 177, 192, 200, 207, 208, 283  
     definizione di, 48  
**wait3** (funzione), 51  
     definizione di, 51  
**wait4** (funzione), 51, 172, 284  
     definizione di, 51  
**WAIT\_ANY** (costante), 49  
**WAIT\_MYPGRP** (costante), 49  
**waitpid** (funzione), 39, 47–51, 177, 192, 207, 208, 224, 227  
     definizione di, 49  
**WCOREDUMP(s)** (macro), 50  
**WEOF** (costante), 141  
**WEXITSTATUS(s)** (macro), 50  
**which** (funzione), 204, 205  
**WIFEXITED(s)** (macro), 50  
**WIFSIGNALED(s)** (macro), 50  
**WIFSTOPPED(s)** (macro), 50  
**WNOHANG** (costante), 49, 208  
**write** (funzione), 102, 117–120, 123, 124, 126, 133–135, 137, 151, 235, 258, 267, 277, 386–388, 392, 397, 400, 410, 426, 460, 461, 466–468, 476  
     definizione di, 123  
**WriteMess** (funzione), 282  
**writev** (funzione), 461  
     definizione di, 261  
**WSTOPSIG(s)** (macro), 50  
**WTERMSIG(s)** (macro), 50  
**WUNTRACED** (costante), 49, 50, 224

**X\_OK** (costante), 109  
**XCASE** (costante), 241

**zombie**, 47, 48, 50, 62, 207, 208, 403

# Bibliografia

- [1] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Prentice Hall PTR, 1995.
- [2] W. R. Stevens, *UNIX Network Programming, volume 1*. Prentice Hall PTR, 1998.
- [3] S. L. R. M. S. R. M. A. Oram and U. Drepper, *The GNU C Library Reference Manual*. Free Software Foundation, 1998.
- [4] A. Rubini and J. Corbet, *Linux Device Driver*. O'Reilly, 2002.
- [5] Aleph1, “Smashing the stack for fun and profit,” *Phrack*, 1996.
- [6] V. Paxson, *Flex, version 2.5*. Free Software Foundation, 1995.
- [7] C. Donnelly and R. M. Stallman, *Bison, the YACC-compatible parser generator*. Free Software Foundation, 2002.
- [8] S. Oullaine, *Practical C*. O'Reilly, 2002.
- [9] A. Gierth, “Unix programming frequently asked questions.”
- [10] D. A. Rusling, *The Linux Kernel*. Linux Documentation Project, 1998.
- [11] W. R. Stevens, *UNIX Network Programming, volume 2*. Prentice Hall PTR, 1998.
- [12] W. R. Stevens, *TCP/IP Illustrated, Volume 1, the protocols*. Addison Wesley, 1994.
- [13] S. Piccardi, *Amministrare GNU/Linux*. Truelite Srl, 2004.
- [14] C. Liu and P. Albitz, *DNS and BIND*. O'Reilly, 1998.