

# UNIVERSITÀ DEL SALENTO

Dipartimento di Ingegneria dell’Innovazione

Corso di Laurea Magistrale in Ingegneria Informatica

---

## PROGRAMMAZIONE DI SISTEMA

*Prof. Franco Tommasi*

# Indice

<b>1 Concetti Fondamentali</b>	<b>10</b>
1.1 Il sistema operativo o kernel . . . . .	10
1.2 Gestione degli errori . . . . .	11
1.2.1 Esempio 1.8 . . . . .	12
<b>2 UNIX Standardizzazione e Implementazioni</b>	<b>13</b>
2.1 Single UNIX Specification . . . . .	13
2.2 Limiti . . . . .	14
2.2.1 sysconf, pathconf, and fpathconf Functions . . . . .	14
2.2.2 Esempio 2.14 . . . . .	15
2.3 Feature Test Macros . . . . .	16
<b>3 Compilazione di un file sorgente C</b>	<b>17</b>
3.1 Introduzione . . . . .	17
3.1.1 Debug . . . . .	19
3.2 Prima compilazione . . . . .	20
<b>4 Makefile</b>	<b>22</b>
4.1 Cos'è il Makefile . . . . .	22
4.2 Comando 'make' . . . . .	22
4.3 Struttura del Makefile . . . . .	22
<b>5 Librerie</b>	<b>25</b>
5.1 Librerie statiche e dinamiche . . . . .	25

5.2	Comandi utili per le librerie . . . . .	26
5.2.1	Librerie statiche (Linux e MacOS) . . . . .	26
5.2.2	Librerie dinamiche (Linux) . . . . .	26
5.2.3	Librerie dinamiche (MacOS) . . . . .	27
5.2.4	Uso delle librerie dinamiche . . . . .	27
<b>6</b>	<b>Input e Output</b>	<b>28</b>
6.1	File Descriptors . . . . .	29
6.2	Esempio 1.4 . . . . .	29
6.3	Funzioni open e openat . . . . .	30
6.4	Funzione creat . . . . .	31
6.5	Funzione close . . . . .	32
6.6	Funzione lseek . . . . .	32
6.6.1	Esempio 3.1 . . . . .	33
6.6.2	Esempio 3.2 . . . . .	34
6.7	Funzione read . . . . .	36
6.8	Funzione write . . . . .	36
6.9	File sharing . . . . .	36
6.10	Operazioni Atomiche . . . . .	37
6.10.1	Funzioni pread() e pwrite() . . . . .	37
6.11	Funzioni dup e dup2 . . . . .	38
6.12	Funzione fcntl . . . . .	39
6.12.1	Esercizio . . . . .	40
6.12.2	Esempio 3.11 . . . . .	41
6.12.3	Esempio 3.12 . . . . .	42
6.13	Funzione ioctl . . . . .	43
6.14	Funzioni link, linkat, unlink, unlinkat e remove . . . . .	44
6.14.1	Esempio 4.16 . . . . .	45
6.15	Funzioni rename e renameat . . . . .	46
6.16	Creare e leggere Links simbolici . . . . .	46

<b>7 Files e Directories</b>	<b>48</b>
7.1 Filesystem . . . . .	48
7.2 Working directory . . . . .	48
7.2.1 Esempio 1.3 . . . . .	48
7.3 Funzione stat, fstat, fstatat e lstat . . . . .	49
7.4 Tipi di file . . . . .	50
7.4.1 Esempio 4.3 . . . . .	51
7.5 Set-User-ID e Set-Group-ID . . . . .	52
7.6 Funzioni access e faccessat . . . . .	52
7.6.1 Esempio 4.8 . . . . .	53
7.7 Funzione umask . . . . .	53
7.7.1 Esempio 4.9 . . . . .	54
7.8 Funzioni chmod, fchmod, fchmodat . . . . .	55
7.8.1 Esempio 4.12 . . . . .	55
7.9 Troncamento di un file . . . . .	56
7.10 Funzioni futimens, utimensat e utimes . . . . .	56
7.10.1 Esempio 4.21 . . . . .	58
7.11 Funzioni mkdir, mkdirat e mkdir . . . . .	58
7.12 Leggere directories . . . . .	59
7.12.1 Esempio 4.22 . . . . .	60
7.13 Funzioni chdir, fchdir e getcwd . . . . .	65
7.14 Funzioni time . . . . .	65
<b>8 Programmi e Processi</b>	<b>70</b>
8.1 Esempio 1.6 . . . . .	70
8.2 Esempio 1.7 . . . . .	71
8.3 Funzione exit . . . . .	73
8.3.1 Funzione atexit . . . . .	73
8.3.2 Esempio 7.3 . . . . .	73
8.4 Variabili d'ambiente . . . . .	74

8.4.1	File memory_dump.c . . . . .	75
8.5	Allocazione di memoria . . . . .	80
8.6	Funzioni getrlimit e setrlimit . . . . .	80
8.6.1	Esempio 7.16 . . . . .	81
8.7	Funzione fork . . . . .	82
8.8	Funzioni wait e waitpid . . . . .	83
8.8.1	Esempio 8.5 . . . . .	84
8.8.2	Esempio 8.6 . . . . .	84
8.8.3	Esempio 8.8 . . . . .	85
8.9	Race Conditions . . . . .	86
8.9.1	Esempio 8.12 . . . . .	87
8.9.2	Esempio 8.13 . . . . .	87
8.10	Funzioni exec . . . . .	89
8.10.1	Esempio 8.16 . . . . .	90
8.11	Funzioni setuid e setgid . . . . .	91
8.12	Funzioni setreuid e setregid . . . . .	92
8.13	Funzioni seteuid e setegid . . . . .	92
8.14	Funzione system . . . . .	92
8.15	Process Times . . . . .	93
8.15.1	Esempio 8.31 . . . . .	94
<b>9</b>	<b>Segnali</b> . . . . .	<b>96</b>
9.1	Esempio 1.10 . . . . .	97
9.2	Funzione signal . . . . .	98
9.2.1	Esempio 10.2 . . . . .	98
9.3	Interrupted System Calls . . . . .	99
9.4	Funzioni Reentrant . . . . .	100
9.5	Reliable-Signal Terminology and Semantics . . . . .	100
9.6	Funzioni kill e raise . . . . .	101
9.7	Funzioni alarm e pause . . . . .	101

9.7.1	Esempio . . . . .	102
9.7.2	Esempio 10.7 . . . . .	103
9.8	Signal Sets . . . . .	103
9.9	Funzione sigprocmask . . . . .	104
9.9.1	Esempio 10.14 . . . . .	105
9.10	Funzione sigpending . . . . .	106
9.10.1	Esempio 10.15 . . . . .	106
9.11	Funzione sigaction . . . . .	108
9.11.1	Esempio sigusr1.c . . . . .	108
9.12	Funzione sigsuspend . . . . .	109
9.12.1	Esempio 10.22 . . . . .	110
9.12.2	Esempio 10.23 . . . . .	112
9.12.3	Esempio 10.24 . . . . .	113
9.13	Nomi e Numeri del segnale . . . . .	114
<b>10 Threads</b>		<b>115</b>
10.1	Thread Identification . . . . .	116
10.2	Thread Creation . . . . .	116
10.2.1	Esempio 11.2 . . . . .	117
10.3	Thread Termination . . . . .	118
10.3.1	Esempio 11.3 . . . . .	119
10.3.2	Esempio thread_incd.c . . . . .	120
10.4	Thread Synchronization . . . . .	121
10.4.1	Mutexes . . . . .	121
	Esempio thread_incr_mutex.c . . . . .	123
10.4.2	Reader-Writer Locks . . . . .	124
10.4.3	Reader-Writer Locling with Timeouts . . . . .	125
10.4.4	Condition Variables . . . . .	125
	Esempio prod_no_condvar.c . . . . .	127
	Esempio prod_condvar.c . . . . .	129

10.5 Thread Attributes . . . . .	131
10.6 Reentrancy . . . . .	133
10.7 Thread-Specific Data . . . . .	134
10.7.1 Esempio ATEST22TLS.c . . . . .	135
10.8 Cancel Options . . . . .	137
10.9 Thread e Segnali . . . . .	138
10.9.1 Esempio 12.16 . . . . .	139
10.10 Thread e I/O . . . . .	141
<b>11 Daemon Processes</b>	<b>142</b>
11.1 Caratteristiche dei Daemon . . . . .	142
11.2 Coding Rules . . . . .	143
11.2.1 Esempio 13.1 . . . . .	144
11.3 Error Logging . . . . .	145
11.4 Single-Instance Daemons . . . . .	147
11.4.1 Esempio 13.6 . . . . .	147
11.5 Daemon Conventions . . . . .	149
11.5.1 Esempio 13.7 . . . . .	149
<b>12 Advanced I/O</b>	<b>152</b>
12.1 Nonblocking I/O . . . . .	152
12.1.1 Esempio 14.1 + 3.12 . . . . .	153
12.2 Record Locking . . . . .	154
12.2.1 fcntl Record Locking . . . . .	154
12.2.2 Esempio lock2.c . . . . .	155
12.2.3 Esempio lock3.c . . . . .	156
12.2.4 Esempio lock4.c . . . . .	158
12.2.5 Esempio lock5.c . . . . .	159
12.3 I/O Multiplexing . . . . .	161
12.3.1 Funzione select() . . . . .	161

Esempio Server echo con select . . . . .	162
12.3.2 Funzione pselect() . . . . .	162
Esempio Server echo con pselect . . . . .	163
12.3.3 Funzione poll() . . . . .	163
Esempio Server echo con poll . . . . .	164
12.3.4 Funzione epoll_create() (solo LINUX) . . . . .	164
<b>13 Interprocess Comunication</b>	<b>166</b>
13.1 Pipe . . . . .	166
13.1.1 Esempio pipe2.c . . . . .	169
13.1.2 Esempio pipe3.c . . . . .	170
13.1.3 Esempio pipe4.c . . . . .	170
13.1.4 Esempio pipe5.c . . . . .	171
13.1.5 Esempio pipe_ls_wc.c . . . . .	172
13.2 FIFOs . . . . .	174
13.3 XSI IPC . . . . .	175
13.4 Semaphores . . . . .	176
13.4.1 Esempio psem_create.c . . . . .	180
13.4.2 Esempio sem1.c . . . . .	182
13.5 Shared Memory . . . . .	184
13.5.1 Esempio shm1.c . . . . .	185
13.5.2 Esempio shm2.c . . . . .	187
<b>14 UNIX Network Programming</b>	<b>189</b>
14.1 Sockets Programming . . . . .	189
14.2 Socket creation . . . . .	191
14.3 Socket Adresses . . . . .	192
14.3.1 Esempio SocketAdress.c . . . . .	194
Inizializzazione di una struttura sockaddr_in (IPv4) . . . . .	195
Stampa di sockaddr_in (IPv4) . . . . .	195

Inizializzazione di una struttura sockaddr_in6 (IPv6) . . . . .	196
Stampa sockaddr_in6 (IPv6) . . . . .	196
14.4 TCP Client e TCP Server . . . . .	197
14.4.1 Funzione bind() . . . . .	197
14.4.2 Funzione listen() . . . . .	198
14.4.3 Funzione accept() . . . . .	198
14.4.4 Funzione connect() . . . . .	198
14.4.5 Funzioni send() e recv() . . . . .	199
14.4.6 Funzione close() . . . . .	200
14.4.7 Local e Romote Adress . . . . .	200
14.4.8 Esempio TCPServer.c . . . . .	200
14.4.9 Esempio TCPClient.c . . . . .	200
14.5 UDP Client e UDP Server . . . . .	200
14.5.1 Funzione sendto() . . . . .	201
14.5.2 Funzione recvfrom() . . . . .	202
14.5.3 Esempio UDPServer.c . . . . .	202
14.5.4 Esempio UDPClient.c . . . . .	202
14.6 Esempio showip.c . . . . .	202
14.7 Esempio server.c . . . . .	202
14.8 Hostname Resolution . . . . .	203
14.9 Funzione getaddrinfo() . . . . .	203
14.10Funzione freeaddrinfo() . . . . .	205
14.11Funzione getnameinfo() . . . . .	205
14.12Funzione gai_strerror() . . . . .	206

# Capitolo 1

## Concetti Fondamentali

### 1.1 Il sistema operativo o kernel

Sistema operativo è quel programma essenziale, sempre in esecuzione in una macchina accesa, grazie al quale possiamo fruire dei servizi di un computer. Questo programma è ovviamente eseguito dal/ dai processore/i del computer e tutti i processori hanno almeno due modalità di funzionamento: il kernel mode e lo user mode. Il kernel mode (detto anche supervisor mode) ha la proprietà di poter eseguire qualsiasi istruzione prevista dal processore e di poter accedere a qualsiasi risorsa della macchina, mentre allo user mode è riservato un sottoinsieme di tutte le istruzioni possibili e un accesso limitato alle risorse. Il sistema operativo di un computer è l'unico programma che viene sempre eseguito in kernel mode.



Il kernel riceve le richieste degli utenti (che in questo caso sono le applicazioni eseguite per conto degli utenti del computer) e le soddisfa in modo ordinato, massimizzando l'efficienza dell'esecuzione e impedendo che un utente danneggi i dati di proprietà di un altro utente o violi la sua privacy ecc. Nel linguaggio dei sistemi operativi, le richieste che le applicazioni fanno al kernel sono dette system call. Una System Call (in italiano “chiamata di sistema”), abbreviata in Syscall, è un metodo utilizzato dai programmi applicativi per comunicare con il core del sistema. Questo metodo viene utilizzato nei moderni sistemi operativi quando un'applicazione o un processo utente deve trasmettere informazioni all'hardware, ad altri processi o al kernel stesso o visionare informazioni da queste fonti. Questo tipo di chiamata è pertanto l'anello di congiunzione tra la modalità utente (user mode) e la modalità kernel (kernel mode).

A tal proposito molto importante è il comando '**man**' che permette di visualizzare le pagine di manuale. Le varie pagine man sono raggruppate in sezioni omogenee per tipo di argomento trattato (ad esempio vi è una sezione per i comandi utente, una per le chiamate di sistema (2), una per i formati dei file di configurazione, ed altre ancora).

## 1.2 Gestione degli errori

Quando si verifica un errore in una delle funzioni del sistema UNIX, spesso viene restituito un valore negativo e l'intero **errno** è solitamente impostato su un valore che indica il motivo. Per esempio, la funzione open restituisce un descrittore di file non negativo se tutto è OK o negativo se si verifica un errore. Il file <errno.h> definisce il simbolo errno e le costanti per ogni valore che errno posso supporre. Ognuna di queste costanti inizia con il carattere E. Inoltre, la prima pagina della Sezione 2 dei manuali del sistema UNIX, denominata intro(2), di solito elenca tutti questi costanti di errore. Ad esempio, se errno è uguale alla costante EACCES, ciò indica un problema di autorizzazione, ad esempio autorizzazione insufficiente per aprire il file richiesto. Il simbolo errno Può essere un numero intero che contiene il numero di errore o una funzione che restituisce un puntatore al numero di errore. La sintassi è:

```
extern int errno;
```

In particolare la variabile **extern** viene utilizzata quando un particolare file deve accedere a una variabile da un altro file. La sintassi per definire una variabile extern in C consiste semplicemente nell'utilizzare la parola chiave **extern** prima della dichiarazione della variabile.

Gli errori definiti in <errno.h> possono essere suddivisi in due categorie: fatali e non fatali. Un errore irreversibile non prevede alcuna azione di ripristino. Gli errori non fatali, invece, possono talvolta essere affrontati in modo più deciso. La maggior parte degli errori non fatali sono temporanei e potrebbe non verificarsi quando c'è meno attività sul sistema.

### 1.2.1 Esempio 1.8

---

```
#include "apue.h"
#include <errno.h>

int
main(int argc, char *argv[])
{
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));
    errno = ENOENT;
    perror(argv[0]);
    exit(0);
}
```

---

Figure 1.8 Demonstrate **strerror** and **perror**

Questo programma C mostra la gestione degli errori utilizzando le funzioni **strerror** e **perror** per visualizzare i messaggi di errore associati a codici di errore specifici:

- **strerror(EACCES)**: Restituisce una stringa descrittiva dell'errore associato al codice di errore EACCES (permesso negato). In questo caso, stampa l'errore su stderr.
- **errno = ENOENT**: Assegna il codice di errore ENOENT (file o directory non trovati) alla variabile globale **errno**.
- **perror(argv[0])**: Stampa un messaggio di errore su stderr che inizia con il nome del programma (preso da **argv[0]**), seguito dal messaggio associato all'errore corrente in **errno**.

# Capitolo 2

## UNIX Standardizzazione e Implementazioni

### 2.1 Single UNIX Specification

Single UNIX Specification (in acronimo SUS) è una definizione collettiva di una serie di standard che i sistemi operativi per computer devono rispettare per potersi fregiare del nome "Unix". La SUS viene sviluppata e mantenuta da Austin Group, ed è basata su specifiche precedenti dell'IEEE e di The Open Group. La SUS è il prodotto scaturito da un progetto nato nella prima metà degli anni ottanta per uniformare l'interfaccia del sistema operativo dal punto di vista dei software progettati per essere eseguiti sulle diverse varianti del sistema operativo Unix. La necessità di una standardizzazione si fece sentire poiché le aziende che avevano acquistato dei computer volevano essere in grado di sviluppare programmi che potevano essere utilizzati su sistemi di produttori diversi senza che fosse necessario reimplementarli da zero. Unix fu scelto come base di un'interfaccia di sistema standard poiché non era legato a nessun produttore in particolare. Questi standard divennero nel 1988 IEEE 1003 (registrato anche come ISO/IEC 9945), o POSIX, che significa proprio Interfaccia di Sistema Operativo Portabile per uniX. In particolare durante il corso realizzeremo programmi per SUSv3.

## 2.2 Limiti

Le implementazioni definiscono molti numeri magici e costanti che possono essere definiti per mezzo di alcuni metodi sviluppati nel tempo. Sono necessari due tipi di limiti:

- Limiti del tempo di compilazione (ad esempio, qual è il valore più grande di un short integer?)
- Limiti di runtime (ad esempio, quanti byte in un filename?)

I limiti del tempo di compilazione possono essere definiti nelle intestazioni che qualsiasi programma può includere durante la compilazione. Ma i limiti di runtime richiedono che il processo chiami una funzione per ottenere il valore del limite. Inoltre, è possibile fissare alcuni limiti su una determinata implementazione, quindi essere definito staticamente in un'intestazione, ma variare in un'altra implementazione e ciò richiederebbe una chiamata di funzione runtime. Un esempio di questo tipo di limite è il massimo numero di byte in un file.

### 2.2.1 sysconf, pathconf, and fpathconf Functions

Alcuni di questi limiti di cui abbiamo parlato precedentemente potrebbero essere disponibili in fase di compilazione, mentre altri devono essere determinati in fase di esecuzione ed in particolare questi ultimi (runtime limits) si ottengono chiamando una delle tre funzioni seguenti:

```
#include <unistd.h>

long sysconf (int name);

long pathconf (const char *pathname, int name);

long fpathconf (int fd, int name);
```

La differenza tra le ultime due funzioni è che una prende come argomento un pathname, mentre l'altro accetta come argomento un file descriptor. Tutte le funzioni richiedono come argomento un *name*, importante per specificare cosa vogliamo sapere.

Dobbiamo esaminare più in dettaglio i diversi valori di ritorno di questi tre funzioni:

- Tutte e tre le funzioni restituiscono  $-1$  e impostano errno su EINVAL se il *name* non è una delle costanti appropriate

- Alcuni nomi possono restituire il valore della variabile (un valore  $\geq 0$ ) o un' indicazione che il valore è indeterminato. Viene indicato un valore indeterminato restituendo  $-1$  e non modificando il valore di errno.

### 2.2.2 Esempio 2.14

Supponiamo di voler realizzare un programma che sia in grado di stampare tutti i valori per ogni simbolo di *pathconf* e *sysconf* leggendo due file di input (*pathconf.sym* e *sysconf.sym*) che contengono elenchi del nome e del simbolo del limite, separati da tabulazioni come ad esempio:

```
NAME_MAX      _PC_NAME_MAX
```

Non tutti i simboli sono definiti su ogni piattaforma, quindi il programma circonda ogni chiamata a *pathconf* e *sysconf* con le necessarie istruzioni `#ifdef`. Tale programma tarsforma la riga come vista prima, nel seguente modo:

---

```
#include "apue.h"
#include <errno.h>
#include <limits.h>

static void pr_sysconf(char *, int);
static void pr_pathconf(char *, char *, int);

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <dirname>");

#ifdef ARG_MAX
    printf("ARG_MAX defined to be %ld\n", (long)ARG_MAX+0);
#else
    printf("no symbol for ARG_MAX\n");
#endif
#ifdef _SC_ARG_MAX
    pr_sysconf("ARG_MAX =", _SC_ARG_MAX);
#else
    printf("no symbol for _SC_ARG_MAX\n");
#endif

/* similar processing for all the rest of the sysconf symbols... */

#ifdef MAX_CANON
    printf("MAX_CANON defined to be %ld\n", (long)MAX_CANON+0);
#else
    printf("no symbol for MAX_CANON\n");
#endif
#ifdef _PC_MAX_CANON
    pr_pathconf("MAX_CANON =", argv[1], _PC_MAX_CANON);
#else
    printf("no symbol for _PC_MAX_CANON\n");
#endif

/* similar processing for all the rest of the pathconf symbols... */
    exit(0);
}
```

## 2.3 Feature Test Macros

Le feature test macros sono fondamentali per scrivere codice portabile e mantenibile, permettendo di evitare errori dovuti a incompatibilità con versioni diverse di standard del linguaggio o di librerie, e per garantire che il software funzioni su diverse piattaforme e compilatori.

Esempi di feature test macros:

- Standard del linguaggio:
  - `__STDC__`: Definito se il compilatore aderisce allo standard C.
  - `__cplusplus`: Definito quando si utilizza un compilatore C++.
- Versione dello standard C o C++:
  - `__STDC_VERSION__`: Definisce la versione dello standard C supportata
  - `__cplusplus`: Indica la versione dello standard C++
- Librerie o funzioni di sistema:
  - `_POSIX_VERSION`: Verifica se sono disponibili caratteristiche conformi allo standard POSIX
  - `_XOPEN_SOURCE`: Macro che abilita funzioni definite nello standard X/Open e POSIX

Ad esempio facendo riferimento al comando gcc per la compilazione di un file sorgente C, possiamo notare la presenza di:

- `-DLINUX` o `-DMACOS`: Questo può essere utile in programmi che devono distinguere tra diversi sistemi operativi (ad esempio, Linux rispetto a Windows o macOS) e attivare porzioni di codice specifiche per il sistema Linux Mac OS
- `-D_GNU_SOURCE` o `-D_DARWIN_SOURCE`: la prima abilita tutte le estensioni GNU disponibili nella GNU C Library (glibc) ed specifico per Linux, mentre la seconda macro, specifica per Mac OS, può essere utilizzata per abilitare funzionalità o estensioni particolari di Darwin.

# Capitolo 3

## Compilazione di un file sorgente C

### 3.1 Introduzione

Un **file sorgente** è un file di testo contenente una serie di istruzioni (dette codice sorgente) scritte in un linguaggio di programmazione (normalmente ad alto livello) pronto per essere trasformato da un compilatore in un programma eseguibile o per essere interpretato da un interprete.

Come già detto un programma in C può essere scritto anche con un semplice editor di testi come il Blocco note di Windows (o simile). Per poter vedere il risultato del codice dobbiamo salvare necessariamente il file con un'estensione ".c", ad esempio "hello.c". A questo punto non dobbiamo fare altro che compilare il programma.

In realtà la compilazione di un programma in C (spesso integrata in ambienti di sviluppo e quindi apparentemente trasparente ed uniforme) segue varie fasi:

- Il codice sorgente viene controllato dal preprocessore che ha i seguenti compiti:
  - rimuovere eventuali commenti presenti nel sorgente;
  - interpretare speciali direttive per il preprocessore denotate da "#", come #include o #define
  - controllare eventuali errori del codice

- Il risultato del preprocessore sarà un nuovo codice sorgente "pulito" ed "espanso" che viene tradotto dal compilatore C in codice assembly; L'assembler sarà incaricato di creare il codice oggetto salvandolo in un file (.o sotto Unix/Linux e .obj in Dos/Windows);
- Il Link editor ha il compito di collegare tutti i file oggetto risolvendo eventuali dipendenze e creando il programma (che non è altro che un file eseguibile).

Per compilare basta posizionarsi nella solita cartella dove risiede il file sorgente e scrivere:

```
gcc -ansi -I../include -Wall -DLINUX -D_GNU_SOURCE ls1.c -o lsl -L../lib -lapue
gcc -ansi -I../include -Wall -DMACOS -D_DARWIN_SOURCE ls1.c -o lsl -L../lib -lapue
```

dove:

- **gcc:** È il compilatore GNU per il linguaggio C
- **-ansi:** Abilita la conformità allo standard ANSI C (C89), disabilitando le estensioni GNU.
- **-I../include:** Aggiunge la directory ../include al percorso di ricerca per i file di intestazione (header files).
- **-Wall:** Abilita tutti gli avvisi di compilazione per aiutare a identificare potenziali problemi nel codice.
- **ls1.c:** È il file sorgente C da compilare
- **-o lsl:** Specifica il nome dell'eseguibile risultante (lsl)
- **-L../lib:** Aggiunge ../lib al percorso di ricerca per le librerie durante il collegamento.
- **-lapue:** Collega la libreria apue (Advanced Programming in the UNIX Environment) presente nella directory specificata da -L.

Per semplicità possiamo scrivere in un file .sh (es. comp.sh) con il seguente comando:

```
gcc -ansi -I../include -Wall -DMACOS -D_DARWIN_SOURCE ${1}.c -o ${1} -L../lib -lapue
```

In modo da poter compilare un file .c in questo modo:

```
./comp.sh file.c
```

### 3.1.1 Debug

Per avviare un processo di debug su Linux, è comune utilizzare strumenti come gdb (GNU Debugger) per il debug di programmi scritti in C o C++.

**Compilare il programma con informazioni di debug:** Prima di eseguire il debug, assicurati di compilare il programma con il **flag -g**:

```
gcc -ansi -g -I../include -Wall -DLINUX -D_GNU_SOURCE ls1.c -o ls1 -L../lib -lapue
```

**Avviare gdb:** Puoi avviare il debug del programma con il seguente comando:

```
gdb ./ls1
```

**Comandi utili di gdb:** Una volta all'interno dell'interfaccia di gdb, puoi utilizzare i seguenti comandi per navigare e controllare il processo di debug:

- **run:** Esegue il programma
- **break <nome\_file>:<numero\_linea>:** Imposta un breakpoint a una linea specifica (es. break main.c:10)
- **break <nome\_funzione>:** Imposta un breakpoint all'inizio di una funzione
- **continue (o c):** Continua l'esecuzione fino al prossimo breakpoint
- **step (o s):** Esegue il programma riga per riga, entrando nelle funzioni
- **next (o n):** Esegue il programma riga per riga senza entrare nelle funzioni
- **print <variabile>:** Visualizza il valore di una variabile
- **bt (backtrace):** Mostra lo stack trace per vedere la sequenza delle chiamate di funzione
- **quit:** Esce da gdb

## 3.2 Prima compilazione

Di seguito la prima compilazione che è stata spiegata a lezione:

**gcc -c bill.c**

Il comando `gcc -c bill.c` è utilizzato per compilare il file sorgente C `fred.c` generando un file oggetto:

- `gcc`: Questo è il compilatore GNU C, utilizzato per compilare programmi scritti in C
- `-c`: Questa opzione indica al compilatore di compilare solo il file sorgente specificato senza procedere al collegamento finale. Questo significa che verrà prodotto un file oggetto (.o) invece di un eseguibile
- `bill.c`: Questo è il file sorgente C che si desidera compilare. Contiene il codice sorgente del programma.

**gcc -o program program.c bill.o**

Il comando `gcc -o program program.c bill.o` viene utilizzato per compilare e collegare il file sorgente `program.c` con il file oggetto `bill.o`, creando un eseguibile chiamato `program`:

- `-o program`: Questa opzione specifica il nome dell'eseguibile che verrà creato.
- `program.c`: Questo è il file sorgente C principale che viene compilato.
- `bill.o`: Questo è un file oggetto precedentemente compilato (probabilmente generato da `bill.c`). Viene incluso nel processo di collegamento per essere combinato con il codice di `program.c`

**./program**

Il comando `./program` è utilizzato per eseguire un file eseguibile chiamato `program` nella directory corrente di un sistema operativo basato su Unix (come Linux o macOS).

**ar crv libdummy.a bill.o**

Il comando `ar crv libdummy.a bill.o` viene utilizzato per creare una libreria statica in formato `.a` contenente uno o più file oggetto.

**gcc -o program program.c -L. -ldummy**

Il comando `gcc -o program program.c -L. -ldummy` viene utilizzato per compilare e creare un eseguibile a partire da un file sorgente C (`program.c`), collegando una libreria statica chiamata `libdummy.a` (Il prefisso `lib` e l'estensione `.a` vengono omessi quando si utilizza l'opzione `-l`).

# Capitolo 4

## Makefile

### 4.1 Cos'è il Makefile

In ogni directory che contiene sorgenti è necessario creare un file di testo, chiamato Makefile, che contiene le istruzioni per la compilazione, quindi specifica i comandi da eseguire, le dipendenze tra i file e l'ordine di esecuzione.

Una volta creato il Makefile è sufficiente lanciare il comando *make* (o *make* obiettivo) per avviare la compilazione, dove obiettivo è il file da creare.

### 4.2 Comando 'make'

Questo comando automatizza il processo di compilazione di programmi complessi che coinvolgono più file sorgente e dipendenze.

In particolare l'opzione *-i* dice al comando *make* di ignorare tutti gli errori che si verificano durante l'esecuzione dei comandi all'interno delle regole.

Quindi il comando *make*, basandosi sul Makefile, controlla se qualcuno fra i prerequisiti è stato modificato più recentemente dell'obiettivo e in caso affermativo esegue i comandi.

### 4.3 Struttura del Makefile

Un semplice Makefile presenta la seguente struttura:

```
obiettivo: prerequisiti
<tab> comando
<tab> comando
<tab> ...
```

dove:

- obiettivo: file da creare
- prerequisiti: file che servono per crearlo
- comando: I comandi (si noti il tab obbligatorio) per creare l'obiettivo.

Vediamo un esempio banale di Makefile:

```
risultato.txt: req1.txt req2.txt req3.txt
    cat req1.txt req2.txt req3.txt > risultato.txt
    cat risultato.txt
    echo $(MAKE)

req1.txt:
    echo tanti saluti > req1.txt

req2.txt:
    echo a casa > req2.txt

req3.txt:
    echo ciao > req3.txt
```

- Il comando '*make*' controlla che il file risultato.txt non esista oppure sia più vecchio dei file req1.txt e req2.txt.
- Quindi esegue il comando cat che crea il file risultato.txt
- Nel caso l'obiettivo sia già aggiornato notifica la cosa con la seguente stringa:

```
make: 'risultato.txt' is up to date
```

Di seguito un esempio complesso di Makefile

```

ROOT=..
PLATFORM=$(shell $(ROOT)/systype.sh)
include $(ROOT)/Make.defines.$(PLATFORM)

PROGS = getputc hello ls1 mycat shell1 shell2 testerror uidgid

all:      $(PROGS)

#es. se c'è ls1: allora fa il pattern
%:      %.c $(LIBAPUE)      #queste istruzioni vengono eseguite per ogni prerequisito di PROGS
      $(CC) $(CFLAGS) $@.c -o $@ $(LDFLAGS) $(LDLIBS)

clean:
      rm -f $(PROGS) $(TEMPFILES) *.o

include $(ROOT)/Make.libapue.inc

```

Figura 4.1: Makefile della directory ‘intro’

- ROOT = ...: Definisce il percorso della directory principale del progetto.
- PLATFORM = (shell(ROOT)/systype.sh): Esegue lo script systype.sh per determinare il tipo di piattaforma (ad esempio, Linux o macOS) e memorizza il risultato nella variabile
- include \$(ROOT)/Make.defines.\$(PLATFORM): Include un file di definizioni specifiche della piattaforma, che potrebbe contenere variabili come il compilatore, flag di compilazione, e altre impostazioni.
- PROGS = getputc hello ls1 mycat shell1 shell2 testerror uidgid: Definisce una lista di programmi che saranno costruiti.
- all: \$(PROGS): La regola principale del Makefile. Quando si esegue make, questa regola indica di costruire tutti i programmi elencati in PROGS.
- La regola di pattern indica che per ogni file sorgente con estensione .c, viene creato un eseguibile con lo stesso nome, dove \$@ rappresenta il target (il nome dell'eseguibile), mentre %.c rappresenta il file sorgente corrispondente

Un’importante osservazione da fare è che se ad esempio vorremmo che questo Makefile prendesse in considerazione un altro programma da noi relizzato, ‘ls2’, allora basterà aggiungerlo alla lista PROGS affinchè anche esso venga compilato nel momento in cui viene eseguito il comando ‘make’ nella directory ‘intro’.

# Capitolo 5

## Librerie

### 5.1 Librerie statiche e dinamiche

Le **librerie statiche** sono archivi di file oggetto che vengono combinati e inclusi direttamente nell'eseguibile finale al momento della compilazione, quindi la dimensione dell'eseguibile sarà maggiore. Le **librerie dinamiche** sono caricate in memoria durante l'esecuzione del programma, non al momento della compilazione, quindi la dimensione dell'eseguibile sarà minore.

La più importante libreria del sistema linux è libc.so, all'interno della quale ci sono le funzioni più utilizzate, e che su macOS è libsystem.dylib. Su linux la libreria lib.c può diventare anche statica utilizzando il flag `-static`, ma questa cosa non è possibile su macOS, tuttavia può essere utilizzata una libreria statica all'interno di un codice

Generalmente le librerie statiche hanno estensione **.a** sia su Linux che MacOS, mentre per quanto riguarda le librerie dinamiche su Linux hanno estensione **.so** (oggetto condiviso) e su MacOS hanno estensione **.dylib**.

Per vedere le librerie condivise che un eseguibile richiede per poter funzionare:

```
ldd nome_eseguibile (Linux)  
otool -L nome_eseguibile (MacOS)
```

## 5.2 Comandi utili per le librerie

### 5.2.1 Librerie statiche (Linux e MacOS)

Comando che compila un file sorgente C (libprova.c) in un file oggetto (libprova.o):

```
gcc -c libprova.c
```

Comando che crea un archivio ovvero una libreria statica (libprova.a) a partire da un file oggetto (libprova.o) che quindi poi viene inserito al suo interno:

```
ar rcs libprova.a libprova.o
```

Comando che compila il file sorgente (useprova.c) in un file oggetto (useprova.o):

```
gcc -Wall -g -c useprova.c -o useprova.o
```

Comando che crea un eseguibile (useprova) a partire dal file oggetto (useprova.o), utilizzando la libreria statica libprova.a:

```
gcc -g -o useprova useprova.o -L. -lprova
```

### 5.2.2 Librerie dinamiche (Linux)

Comando che compila il file sorgente C (libprova.c) in un file oggetto:

```
gcc -fPIC -Wall -g -c libprova.c
```

Comando che crea una libreria condivisa (.so) a partire da un file oggetto (libprova.o):

```
gcc -g -shared -Wl,-soname,libprova.so.0 -o libprova.so.0.0 libprova.o -lc
```

Comando che crea un link simbolico chiamato libprova.so.0 che punta al file libprova.so.0.0:

```
ln -sf libprova.so.0.0 libprova.so.0
```

Comando che crea un link simbolico chiamato libprova.so che punta al file libprova.so.0:

```
ln -sf libprova.so.0 libprova.so
```

### 5.2.3 Librerie dinamiche (MacOS)

Comando che crea una libreria dinamica sui sistemi MacOS:

```
gcc -dynamiclib libprova.c -o libprova.dylib
```

### 5.2.4 Uso delle librerie dinamiche

Comando che compila in file sorgente C (useprova.c) in un file oggetto (.o):

```
gcc -Wall -g -c useprova.c -o useprova.o
```

Comando che compila un programma eseguibile da un file oggetto (useprova.o) collegando una libreria condivisa (libprova.so o libprova.o):

```
gcc -g -o useprova useprova.o -L. -lprova
```

Comando utilizzato per far eseguire il programma specificando la directory corrente perchè su Linux per utilizzare una libreria dinamica che non è di sistema bisogna specificare il path:

```
LD_LIBRARY_PATH='pwd' ./useprova
```

In particolare si può utilizzare *export* per impostare la variabile d'ambiente LD\_LIBRARY\_PATH alla directory corrente e renderla disponibile per i processi eseguiti in quella sessione dello shell

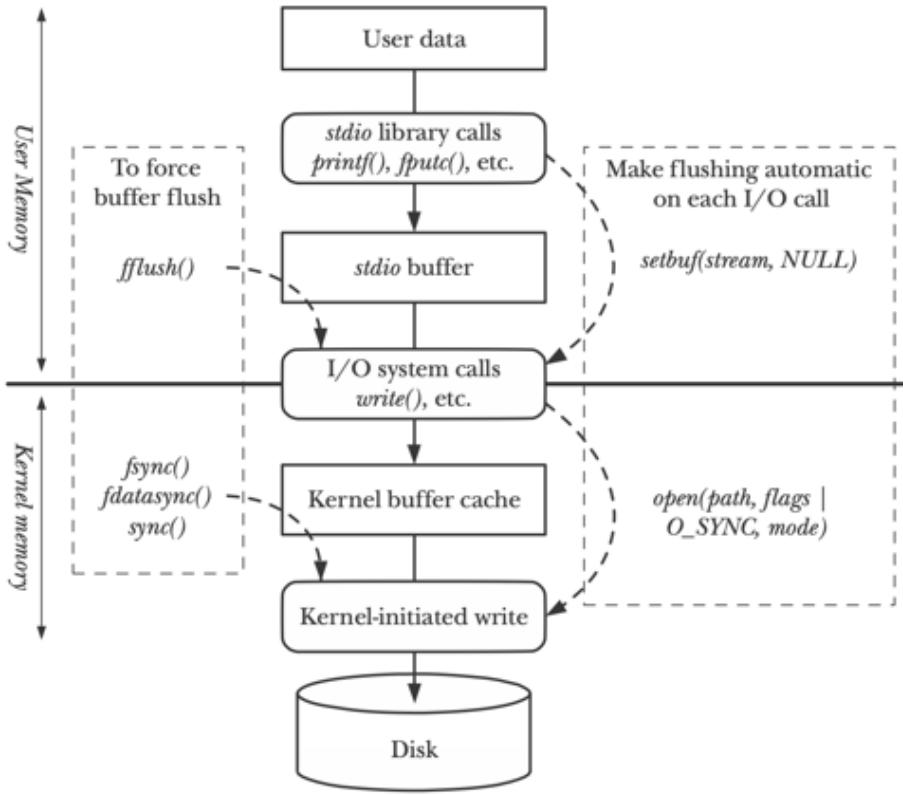
Comando utilizzato per visualizzare le dipendenze delle librerie condivise del programma useprova:

```
LD_LIBRARY_PATH='pwd' ldd useprova
```

# Capitolo 6

## Input e Output

Molti dei file I/O su un sistema Unix possono essere eseguiti usando solo 5 funzioni: open, read, write, lseek e close. Queste funzioni vengono spesso definite come *unbuffered I/O*, dove il termine unbuffered significa che ogni read o write invoca una system call nel kernel. In figura è fornita una panoramica del buffering impiegato (per i file di output) dalla libreria stdio e dal kernel, insieme ai meccanismi per controllare ciascun tipo di buffering. Viaggiando verso il basso attraverso il centro di questo diagramma, vediamo il trasferimento dei dati utente da parte delle funzioni della libreria stdio al buffer stdio, che viene mantenuto nello spazio della memoria utente. Quando questo buffer è pieno, la libreria stdio invoca la chiamata di sistema write(), che trasferisce i dati nel buffer del kernel cache (mantenuta nella memoria del kernel). Alla fine, il kernel avvia un'operazione sul disco per trasferire i dati sul disco. Il lato sinistro della figura mostra le chiamate che possono essere utilizzate in qualsiasi momento per forzare esplicitamente uno svuotamento di uno dei buffer. Il lato destro mostra le chiamate che possono essere utilizzate per rendere automatico lo svuotamento, disabilitando il buffering nella libreria stdio o rendendo sincrone le chiamate del sistema di output del file, in modo che ogni write() venga immediatamente scaricato sul disco.



## 6.1 File Descriptors

I descrittori di file sono normalmente piccoli numeri interi non negativi che il kernel utilizza per identificare i file a cui accede un processo. Ogni volta che apre un file esistente o crea un nuovo file, il kernel restituisce un descrittore di file che utiliziamo quando vogliamo leggere o scrivere il file. Per convenzione, tutte le shell aprono tre descrittori ogni volta che viene eseguito un nuovo programma: Standard Input, Standard Output, and Standard Error.

## 6.2 Esempio 1.4

In questo programma viene dichiarato un buffer di lunghezza 4096, nel quale viene copiato comunque qualsiasi file normale. La funzione di lettura restituisce il numero di byte letti e viene utilizzato questo valore come numero di byte da scrivere. Quando viene raggiunta la fine del file di input, la funzione 'read' restituisce 0 e il programma si ferma. Se si verifica un errore di lettura, read restituisce un valore negativo.

---

```

#include "apue.h"
#define BUFFSIZE    4096
int
main(void)
{
    int      n;
    char    buf[BUFFSIZE];
    while ((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
    if (n < 0)
        err_sys("read error");
    exit(0);
}

```

---

Figure 1.4 Copy standard input to standard output

Molto interessante è l'intestazione `<unistd.h>`, inclusa da `apue.h`, e le due costanti `STDIN_FILENO` e `STDOUT_FILENO` fanno parte dello standard POSIX. Le costanti `STDIN_FILENO` e `STDOUT_FILENO` sono definite in `<unistd.h>` e sono i descrittori di file per l'input standard e l'output standard. Questi valori sono 0 e 1, rispettivamente, come richiesto da POSIX, ma useremo i nomi per la leggibilità. Approfondendo i tre Standard file Descriptor sono:

- **`STDIN_FILENO` (0):** Questo è il descrittore di file per l'input standard. Per impostazione predefinita, è collegato alla tastiera o al flusso di input da cui il programma legge i dati. Il valore è 0.
- **`STDOUT_FILENO` (1):** Questo è il descrittore di file per l'output standard. Per impostazione predefinita, è connesso al terminale o alla console su cui il programma scrive il suo output. Il valore è 1.
- **`STDERR_FILENO` (2):** Questo è il descrittore di file per l'errore standard. Viene utilizzato per l'emissione di messaggi di errore e di diagnostica. Per impostazione predefinita, è anche connesso al terminale o alla console. Il valore è 2.

## 6.3 Funzioni open e openat

Un file viene aperto o creato chiamando la funzione `open` o la funzione `openat`.

```
#include <fcntl.h>
```

```

int open(const char *path, int oflag, ... /*mode_t mode*/);
int open(int fd, const char *path, int oflag, ... /*mode_t mode*/);

```

Ed entrambi ritornano: file descriptor se è OK,  $-1$  se è errore.

E' importante sottolineare che il numero e i tipi dei rimanenti argomenti potrebbero variare e per queste funzioni l'ultimo argomento è usato solo quando viene creato un nuovo file. Il parametro *path* è il nome del file da aprire o creare, mentre il secondo parametro *oflag* specifica l'operazione che vogliamo svolgere attraverso una serie di costanti specificate in <fcntl.h>:

O_RDONLY	Apri solo per lettura
O_WRONLY	Apri solo per scrittura
O_RDWR	Apri per scrittura e lettura
O_APPEND	Aggiungi alla fine del file ad ogni scrittura
O_CREAT	Crea il file se non esiste. Questa opzione richiede un terzo argomento alla funzione open (un quarto argomento della funzione openat) — il mode, che specifica i bit di autorizzazione di accesso del nuovo file
O_NONBLOCK	Se il percorso si riferisce a un FIFO, a un file speciale a blocchi o a un file speciale a caratteri, questa opzione imposta la modalità non bloccante sia per l'apertura del file che per il successivo I/O
O_TRUNC	Se il file esiste e se è stato aperto correttamente per sola scrittura o lettura-scrittura, troncare la sua lunghezza a 0, quindi il suo contenuto viene cancellato

È garantito che il descrittore di file restituito da open e openat sia il descrittore inutilizzato con il numero più basso. Questo fatto viene utilizzato da alcune applicazioni per aprire un nuovo file su input standard, output standard o errore standard. Ad esempio, un'applicazione potrebbe chiudere normalmente l'output standard, descrittore di file 1 e quindi aprire un altro file, sapendo che verrà aperto sul descrittore di file 1.

## 6.4 Funzione creat

Un nuovo file può essere creato richiamando la funzione creat

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```

Restituisce: descrittore di file aperto in sola scrittura se OK, -1 in caso di errore

Questa funzione è equivalente a:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Una carenza di creat è che il file viene aperto solo per la scrittura. Prima che venisse fornita la nuova versione di open, se stavamo creando un file temporaneo che volevamo scrivere e poi rileggere, dovevamo chiamare creat, close e quindi open. Un modo migliore è usare la funzione open in questo modo:

```
open(path, O_RDWR | O_CREAT | O_TRUNC, mode);
```

## 6.5 Funzione close

Un file aperto viene chiuso richiamando la funzione close

```
#include <unistd.h>
int close(int fd);
```

Restituisce: 0 se è OK, -1 in caso di errore

## 6.6 Funzione lseek

L'offset di un file aperto può essere impostato esplicitamente chiamando lseek

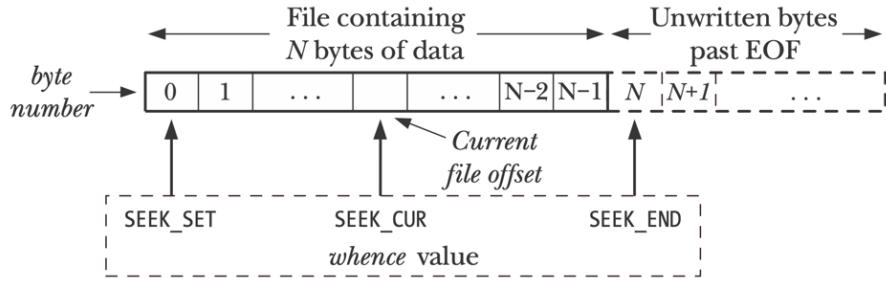
```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Restituisce: nuovo offset del file se OK, -1 in caso di errore.

L'interpretazione dell'offset dipende dal valore dell'argomento *whence*:

- Se *whence* è **SEEK\_SET**, l'offset del file è impostato sui byte di offset dall'inizio del file

- Se *whence* è **SEEK\_CUR**, l'offset del file è impostato sul suo valore corrente più l'offset. L'offset può essere positivo o negativo
- Se *whence* è **SEEK\_END**, l'offset del file è impostato sulla dimensione del file più l'offset. L'offset può essere positivo o negativo.



Poiché una chiamata riuscita a `lseek` restituisce il nuovo offset del file, possiamo cercare zero byte dalla posizione corrente per determinare l'offset corrente:

```
off_t currpos;
currpos = lseek(fd, 0, SEEK_CUR);
```

Ecco alcuni esempi della chiamata `lseek()`:

<code>lseek(fd, 0, SEEK_SET);</code>	Inizio del file
<code>lseek(fd, 0, SEEK_END);</code>	Byte successivo dopo la fine del file
<code>lseek(fd, -1, SEEK_END);</code>	Ultimo byte del file
<code>lseek(fd, -10, SEEK_CUR);</code>	10 bytes precedenti rispetto alla posizione corrente
<code>lseek(fd, 10000, SEEK_END);</code>	10001 bytes dopo l'ultimo byte del file

### 6.6.1 Esempio 3.1

Il seguente programma verifica se è possibile eseguire un'operazione di seek sullo standard input. Invochiamo questo programma compilato che si trova nella dir corrente e che si chiama `seek`, reindirizzando il contenuto del file `hole.c` come input standard del programma stesso:

```
./seek < hole.c
```

---

```
#include "apue.h"

int
main(void)
{
    if (lseek(STDIN_FILENO, 0, SEEK_CUR) == -1)
        printf("cannot seek\n");
    else
        printf("seek OK\n");
    exit(0);
}
```

---

Normalmente, l'offset corrente di un file deve essere un numero intero non negativo. È possibile, tuttavia, alcuni dispositivi potrebbero consentire compensazioni negative, probabilmente perché lo standard input è collegato a un terminale. Ma per i file normali, l'offset deve essere non negativo.

L'offset del file può essere maggiore della dimensione corrente del file, nel qual caso la successiva scrittura sul file estenderà il file. Questa operazione viene definita creazione di un buco in un file ed è consentita. Tutti i byte in un file che non sono stati scritti vengono riletti come 0. In un file, un "hole" (o buco), quindi, si riferisce a una regione vuota di dati che non occupa spazio fisico sul disco, ma che appare logicamente come se esistesse. Un sparse file è un file che contiene regioni di dati vuoti (buchi) che non sono effettivamente memorizzate su disco, ma che vengono interpretate dal file system come una sequenza di zeri.

### 6.6.2 Esempio 3.2

Il seguente programma crea un file con un hole:

---

```
#include "apue.h"
#include <fcntl.h>

char    buf1[] = "abcdefghijkl";
char    buf2[] = "ABCDEFGHIJ";

int
main(void)
{
    int      fd;

    if ((fd = creat("file.hole", FILE_MODE)) < 0)
        err_sys("creat error");

    if (write(fd, buf1, 10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */
```

```

    if (lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");
    /* offset now = 16384 */

    if (write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    exit(0);
}

```

---

Dopo gli include della libreria e delle funzioni di controllo dei file, vengono definiti due array di caratteri (buffer) di 10 caratteri ciascuno. Come già studiato, `creat("file.hole", FILE_MODE)` crea un nuovo file chiamato file.hole con i permessi definiti da FILE\_MODE (di solito 0644, ovvero il proprietario può leggere e scrivere, altri possono solo leggere). La macro FILE\_MODE può essere definita in "apue.h", <fcntl.h> oppure potrebbe trattarsi di una macro di sistema che su Linux può essere trovata con i seguenti comandi:

```

grep -r "#define" /usr/include
grep -rw FILE_MODE /usr/include

```

Mentre su MacOs si trova in questa directory:

```

/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/...
.../SDKs/MacOSX.sdk/usr/include

```

Nel nostro caso si trovava in `apue.3e/include/apue.h`, nel quale c'è scritto:

```
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
```

Come è stato già visto prima, questa funzione può essere sostituita con:

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

La funzione `write(fd, buf1, 10)` scrive i 10 byte del buffer buf1 (cioè "abcdefghijkl") nel file e dopo questa scrittura l'offset sarà 10.

La funzione `lseek(fd, 16384, SEEK_SET)` sposta l'offset di scrittura nel file a 16384, creando un buco (hole) di circa 16 KB tra la scrittura precedente e quella successiva, di conseguenza nessun dato viene scritto tra l'offset 10 e 16384.

La seconda funzione `write` scrive i 10 byte del buffer buf2 (cioè "ABCDEFGHIJ") all'offset 16384.

## 6.7 Funzione read

I dati vengono letti da un file aperto con la funzione read

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t nbytes);
```

Se la lettura ha esito positivo, viene restituito il numero di byte letti. Se la fine del file è incontrato, viene restituito 0.

## 6.8 Funzione write

I dati vengono scritti in un file aperto con la funzione di write

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t nbytes);
```

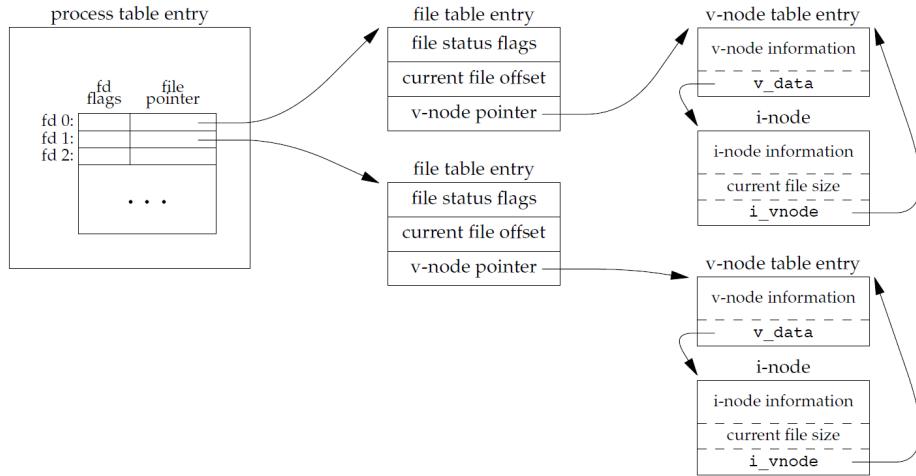
Il valore restituito è solitamente uguale all'argomento nbytes; in caso contrario, si è verificato un errore. Una causa comune di errore di scrittura è il riempimento del disco o il superamento del limite di dimensione del file per un determinato processo.

## 6.9 File sharing

Il kernel utilizza tre strutture dati per rappresentare un file aperto e le relazioni tra di esse determinano l'effetto che un processo ha su un altro per quanto riguarda la condivisione dei file:

- Ogni processo ha una voce nella tabella dei processi, che possiamo pensare come ad un vettore con una entry per file descriptor. Per ogni file descriptor c'è file descriptor flags e un puntatore ad una file table entry.
- Il kernel ha una file table entry per tutti i file aperti ed ognuna di queste contiene i file status flags per il file come read, write, append; il current file offset e un puntatore al v-node table entry del file.

- Ogni file aperto (o dispositivo) ha una struttura v-node che contiene informazioni sul tipo di file e puntatori alle funzioni che operano sul file. Spesso v-node contiene anche l'i-node per il file.



## 6.10 Operazioni Atomiche

Le operazioni atomiche sono operazioni che vengono eseguite in modo indivisibile, ovvero senza la possibilità di essere interrotte o influenzate da altre operazioni. In altre parole, un'operazione atomica è garantita per essere eseguita completamente o per niente, e nessun'altra operazione può osservare uno stato intermedio di quell'operazione.

Con `read()` e `write()`, se più thread accedono contemporaneamente allo stesso file descriptor, il puntatore di file condiviso potrebbe causare sovrapposizioni e corruzione dei dati, mentre `pread()` e `pwrite()` evitano conflitti tra thread perché ognuno può leggere o scrivere a posizioni diverse del file senza modificare il puntatore di file globale.

### 6.10.1 Funzioni `pread()` e `pwrite()`

Le funzioni `pread()` e `pwrite()` sono versioni delle funzioni standard `read()` e `write()`, ma con una differenza fondamentale: permettono di specificare un offset di lettura o scrittura all'interno del

file, senza modificare il puntatore di file condiviso (file pointer) utilizzato dal sistema operativo per tenere traccia della posizione corrente di lettura o scrittura nel file:

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
```

Restituisce: numero di byte letti, 0 se fine del file, -1 in caso di errore.

```
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```

Restituisce: numero di byte scritti, -1 in caso di errore. Chiamare pwrite() equivale a chiamare lseek seguito da una chiamata a write, con simili eccezioni.

## 6.11 Funzioni dup e dup2

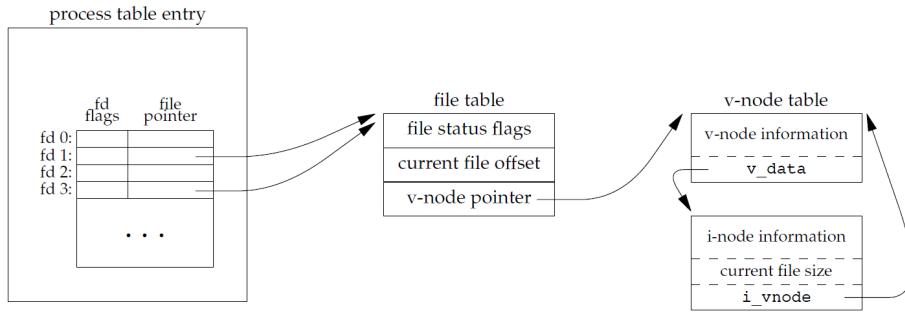
Le funzioni dup() e dup2() sono usate per duplicare file descriptor, ovvero piccoli numeri interi utilizzati dal sistema operativo per rappresentare file aperti.

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```

Entrambi restituiscono: nuovo descrittore di file se OK, -1 in caso di errore.

È garantito che il nuovo descrittore di file restituito da dup sia il descrittore di file disponibile con il numero più basso. Con dup2 specifichiamo il valore del nuovo descrittore con l'argomento fd2 e in particolare se fd2 è già aperto, viene prima chiuso. Se fd è uguale a fd2, allora dup2 restituisce fd2 senza chiuderlo. Quando un file descriptor duplicato (ad esempio, con dup(), dup2()), il flag FD\_CLOEXEC può essere impostato o cancellato. Se è impostato, il file descriptor sarà chiuso automaticamente al momento della chiamata a exec(). Se è cancellato, il file descriptor rimarrà aperto anche dopo l'esecuzione di un nuovo programma.

Il nuovo descrittore di file restituito come valore delle funzioni condivide lo stesso voce della tabella file come argomento fd, come mostrato in figura:



## 6.12 Funzione fcntl

La funzione `fcntl()` viene utilizzata per controllare le proprietà e il comportamento dei file descriptor, quindi permette di modificare le proprietà di un file che è già aperto.

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* int arg */ );
```

Restituisce: dipende da cmd se OK (vedi seguito), `-1` in caso di errore

Il parametro `cmd` della funzione `fcntl()` specifica l'operazione che si desidera eseguire sul file descriptor fornito. Può assumere diversi valori che determinano quale azione viene intrapresa.

I valori di tali parametri sono specificati in `<fcntl.h>`, vediamone alcuni:

- **F\_DUPFD:** consente di duplicare un file descriptor. Il nuovo descrittore di file viene restituito come valore della funzione. È il numero più basso descrittore che non è già aperto e che è maggiore o uguale al terzo argomento. Si differenzia da `dup` perché quest'ultimo duplica un file descriptor assegnando il valore più basso disponibile (indipendentemente dal numero di file descriptor) e anche da `dup2` perché a differenza di quest'ultimo se esiste già un file descriptor, questa non lo chiude ma va avanti fino a quando non ne trova uno libero, quindi prende il primo numero maggiore rispetto a quello già usato.
- **F\_GETFD:** permette di ottenere lo stato dei flag associati a un file descriptor.
- **F\_SETFD:** permette di impostare i flag associati a un file descriptor. Si usa per gestire il flag `FD_CLOEXEC` specifico del file descriptor. Quando il flag `FD_CLOEXEC` è impostato, il file descriptor viene chiuso automaticamente durante una chiamata a una

funzione della famiglia exec. Questo è utile per evitare che file descriptor aperti vengano ereditati dai processi figli.

- **F\_SETFL:** permette di ottenere i flag di accesso (file status flags) associati a un file descriptor. Questi flag controllano il modo in cui il file è stato aperto (ad esempio, in sola lettura, lettura-scrittura, non bloccante, ecc.).
- **F\_GETFL:** recuperare i flag di stato (file status flags) associati a un file descriptor. Questi flag indicano come il file è stato aperto e come viene gestito durante le operazioni di lettura/scrittura.

### 6.12.1 Esercizio

Di seguito il file sorgente di un programma:

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int             fdin, fdout;
    void            *src, *dst;
    size_t          copysz;
    struct stat     sbuf;
    off_t           fsz = 0;
    char buf1[] = "abcdefghijklm";

    if (argc != 2)
        err_quit("usage: %s <fromfile>", argv[0]);

    if ((fdin = open(argv[1], O_RDONLY)) < 0)
        err_sys("can't open %s for reading", argv[1]);

    printf("Waiting for privilage change...")
    sleep(40)

    if ((flags = fcntl(fdin, F_SETFL, O_RDWR)) < 0)
        err_sys("write flag for %s not accepted\n", argv[1]);

    if (write(fdin, buf1, 10) != 10)
        err_sys("buf1 write error");

    exit(0)
}
```

I permessi di un file vengono controllati sempre e solo prima dell'apertura di un file, dunque ora realizziamo un programma all'interno del quale in un primo momento viene aperto un file in sola lettura e, dopo aver modificato i permessi eliminando quello per la scrittura, sfruttando

la flag F\_SETFL si setta il flag in modo che risulti che il file sia stato aperto sia in lettura che in scrittura. Verifichiamo se effettivamente, chiamando la funzione write(), questo avviene senza alcun problema.

Per la compilazione del programma, dopo aver creato un file dummy.txt settando i permessi di scrittura e lettura per l'utente (chmod 600), durante i 40 secondi di sleep sono stati cambiati i permessi del file con il comando chmod 400, mantenendo quindi solo la possibilità di lettura. Al termine della compilazione del programma non è avvenuta alcuna scrittura, quindi questo significa che i permessi sono stati controllati due volte: all'apertura del file in sola lettura e quando è stato utilizzato il flag F\_SETFL.

### 6.12.2 Esempio 3.11

Il programma accetta un singolo argomento della riga di comando che specifica un file descrittore e stampa una descrizione dei flag di file selezionati per quel descrittore utilizzando la system call *fcntl*. Nel seguente programma la funzione fcntl viene chiamata con il comando F\_GETFL per ottenere i flag del descrittore di file. Se fcntl fallisce, l'errore viene segnalato utilizzando err\_sys.

Nell'implementazione dello switch case, *val & O\_ACCMODE* maschera i bit della modalità di accesso e poi li confronta con le possibili modalità:

- O\_RDONLY: Solo lettura.
- O\_WRONLY: Solo scrittura.
- O\_RDWR: Lettura e scrittura.

Se nessuna di queste corrisponde, il programma genera un errore (err\_dump).

Il programma controlla quindi i seguenti flag e stampa i messaggi corrispondenti:

- O\_APPEND: Il file è aperto in modalità append.
- O\_NONBLOCK: La modalità non bloccante è abilitata.
- O\_SYNC: Le scritture sono sincrone (ossia scritte direttamente su disco)

Viene effettuato un controllo speciale per i sistemi che definiscono O\_FSYNC (che potrebbe differire da O\_SYNC).

---

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int      val;
    if (argc != 2)
        err_quit("usage: a.out <descriptor#>");
    if ((val = fcntl(atoi(argv[1]), F_GETFL, 0)) < 0)
        err_sys("fcntl error for fd %d", atoi(argv[1]));
    switch (val & O_ACCMODE) {
    case O_RDONLY:
        printf("read only");
        break;
    case O_WRONLY:
        printf("write only");
        break;
    case O_RDWR:
        printf("read write");
        break;
    default:
        err_dump("unknown access mode");
    }
    if (val & O_APPEND)
        printf(", append");
    if (val & O_NONBLOCK)
        printf(", nonblocking");
    if (val & O_SYNC)
        printf(", synchronous writes");
#ifndef _POSIX_C_SOURCE || !defined(O_FSYNC) || (O_FSYNC != O_SYNC)
    if (val & O_FSYNC)
        printf(", synchronous writes");
#endif
    putchar('\n');
    exit(0);
}
```

---

### 6.12.3 Esempio 3.12

Questa funzione è utile quando si desidera modificare lo stato di un file aperto senza cambiare il modo in cui è stato inizialmente aperto. Ad esempio, potresti voler rendere un file "non bloccante" (O\_NONBLOCK) o abilitarne l'append (O\_APPEND), senza alterare altri flag già esistenti. Non possiamo semplicemente emettere un comando F\_SETFD o F\_SETFL, poiché ciò potrebbe disattivare i bit di flag precedentemente impostati. La funzione set\_fl() prende due argomenti: il descrittore di file che rappresenta il file su cui vogliamo modificare i flag e i nuovi flag che vogliamo impostare sul file. Questi flag sono aggiunti ai flag esistenti del

file. La system call *fcntl(fd, F\_GETFL, 0)* recupera i flag correnti associati al descrittore di file fd. Infine *val |= flags* è un’operazione bit a bit OR ( $|=$ ) che aggiunge i nuovi flag specificati nel parametro flags ai flag esistenti ed in questo modo, dopo aver modificato i flag, la system call *fcntl(fd, F\_SETFL, val)* imposta i nuovi flag per il file descritto da fd.

---

```
#include "apue.h"
#include <fcntl.h>

void
set_fl(int fd, int flags) /* flags are file status flags to turn on */
{
    int      val;

    if ((val = fcntl(fd, F_GETFL, 0)) < 0)
        err_sys("fcntl F_GETFL error");

    val |= flags;          /* turn on flags */

    if (fcntl(fd, F_SETFL, val) < 0)
        err_sys("fcntl F_SETFL error");
}
```

---

## 6.13 Funzione ioctl

La funzione ioctl è sempre stata il punto di riferimento per le operazioni di I/O. Qualunque cosa che non può essere espressa utilizzando una delle altre funzioni già viste (open, close, read, write), può essere specificato con un ioctl.

```
#include <unistd.h> /* System V */
#include <sys/ioctl.h> /* BSD and Linux */

int ioctl(int fd, int request, ...);
```

Restituisce  $-1$  in caso di errore o qualsiasi altra cosa se è OK

Alcuni dei comandi *request* comuni che si possono usare con ioctl() includono:

- **TCGETS:** Ottiene le impostazioni del terminale (struttura termios)
- **TCSETS:** Imposta le nuove configurazioni del terminale
- **FIONREAD:** Ottiene il numero di byte che possono essere letti immediatamente senza bloccare

- **SIOCGIFADDR:** Ottiene l'indirizzo IP di un'interfaccia di rete

Tutti i valori di *request* possono essere consultati nelle pagine di manuale *man ioctl()*

## 6.14 Funzioni link, linkat, unlink, unlinkat e remove

Come sappiamo un file può avere molteplici directory entries pointing al suo i-node, quindi possiamo utilizzare sia il link function che il linkat function per creare un link ad un file già esistente.

```
#include <unistd.h>

int link(const char *existingpath, const char *newpath);
int linkat(intefd, const char *existingpath, int nfd,
           const char *newpath, int flag);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

Queste funzioni creano una nuova directory entry, *newpath*, che fa riferimento al file esistente *existingpath*. Se il *newpath* esiste già, viene restituito un errore. Solo l'ultimo componente del nuovo percorso viene creato, mentre tutto il resto del path deve già esistere.

Con la funzione linkat, il file esistente viene specificato sia da *efd* che dall'argomento *existingpath* e il nuovo nome del percorso è specificato sia da *nfd* che da *newpath* argomenti. Per impostazione predefinita, se uno dei percorsi è relativo, viene valutato rispetto al descrittore di file corrispondente. Se uno dei descrittori di file è impostato su AT\_FDCWD, il percorso corrispondente, se è un percorso relativo, viene valutato rispetto alla corrente directory. Se uno dei due percorsi è assoluto, allora il descrittore di file corrispondente l'argomento viene ignorato.

Per rimuovere una directory entry già esistente utilizziamo:

```
#include <unistd.h>

int unlink(const char *pathname);
int unlinkat(int fd, const char *pathname, int flag);
```

Restituiscono 0 in caso di successo e -1 in caso di errore.

Queste funzioni rimuovono la directory entry e diminuiscono il conteggio dei collegamenti del file a cui fa riferimento il percorso. Se sono presenti altri collegamenti al file, i dati nel file rimangono accessibili tramite gli altri link. Il file non viene modificato se si verifica un errore. Come accennato in precedenza, per rimuovere un link da un file, dobbiamo avere i permessi di scrittura ed esecuzione nella directory contenente la directory entry, poiché è la directory entry che rimuoveremo.

### 6.14.1 Esempio 4.16

Il programma apre un file e dopo rimuove il collegamento ed infine va in modalità sleep per 15 secondi prima di terminare.

---

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    if (open("tempfile", O_RDWR) < 0)
        err_sys("open error");
    if (unlink("tempfile") < 0)
        err_sys("unlink error");
    printf("file unlinked\n");
    sleep(15);
    printf("done\n");
    exit(0);
}
```

---

Questa proprietà di unlink viene spesso utilizzata da un programma per garantire che un file temporaneo creato non verrà lasciato in giro nel caso in cui il programma si arresti in modo anomalo. Il processo crea un file utilizzando open o creat e quindi chiama immediatamente unlink. Il file non viene eliminato, tuttavia, perché è ancora aperto. Solo quando il processo chiude il file o termina, il che fa sì che il kernel chiuda tutti i file aperti, il file viene eliminato. Se il percorso è un collegamento simbolico, lo scollegamento rimuove il collegamento simbolico, non il file a cui fa riferimento il collegamento. Non esiste alcuna funzione per rimuovere il file a cui fa riferimento un link simbolico dato il nome del collegamento.

Possiamo anche effettuare tale scollegamento di un file o di una directory con la funzione *remove* che per un file corrisponde alla funzione *unlink* e per una directory a *rmdir*.

```
#include <stdio.h>

int remove(const char *pathname);
```

Restituisce 0 in caso di successo e  $-1$  in caso di errore

## 6.15 Funzioni rename e renameat

Un file o una directory può essere rinominata con le funzioni rename o renameat

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
int renameat(int oldfd, const char *oldname, int newfd, const char *newname);
```

Restituiscono 0 in caso di successo e  $-1$  in caso di errore

Se *oldname* specifica un file che non è una directory, stiamo rinominando un file o un collegamento simbolico. In questo caso, se *newname* esiste, non può fare riferimento a una directory.

Se *newname* esiste e non è una directory, viene rimosso e *oldname* viene rinominato in *newname*. Dobbiamo avere i permessi di scrittura per la directory che contiene *oldname* e la directory contenente *newname*, poiché stiamo cambiando entrambe le directory.

## 6.16 Creare e leggere Links simbolici

Un link simbolico può essere creato con le funzioni symlink o symlinkat.

```
#include <unistd.h>

int symlink(const char *actualpath, const char *sympath);
int symlinkat(const char *actualpath, int fd, const char *sympath);
```

Restituiscono 0 in caso di successo e  $-1$  in caso di errore

Viene creata una nuova directory entry , *sympath*, che punta a *actualpath*. Non è necessario che *actualpath* esista quando viene creato il link simbolico ed inoltre *actualpath* e *sympath* non devono necessariamente risiedere nello stesso file system.

La funzione symlinkat è simile a symlink, ma l'argomento *sympath* è valutato rispetto alla

directory a cui fa riferimento il descrittore di file aperto per quella directory (specificata dall'argomento *fd*). Se l'argomento *sympath* specifica un percorso assoluto o se l'argomento *fd* ha il valore speciale AT\_FDCWD, allora *symlinkat* si comporta allo stesso modo di *symlink*.

Poiché la funzione *open* segue un collegamento simbolico, abbiamo bisogno di un modo per aprire il collegamento stesso e leggere il nome nel link. Le funzioni *readlink* e *readlinkat* fanno questo:

```
#include <unistd.h>

ssize_t readlink(const char* restrict pathname,
                  char *restrict buf, size_t bufsize);

ssize_t readlinkat(int fd, const char* restrict pathname,
                  char *restrict buf, size_t bufsize);
```

Restituiscono il numero di bytes letti in caso di successo e  $-1$  in caso di errore.

Queste funzioni combinano l'azione di *open*, *read* e *close*. In caso di successo restituiscono il numero di bytes messi nel *buf*. Il comportamento di *readlinkat* è lo stesso di *readlink* se l'argomento *pathname* è un percorso assoluto o quando l'argomento *fd* ha il valore AT\_FDCWD.

# Capitolo 7

## Files e Directories

### 7.1 Filesystem

Una memoria di massa è un supporto fisico che può essere, con apposito software, suddiviso in più supporti logici, detti partizioni, ciascuno dei quali è in grado di ospitare un filesystem (non necessariamente per tutte dello stesso tipo). Un filesystem è essenzialmente un database che permette la memorizzazione di file e l'accesso rapido ai dati degli stessi da parte del sistema e delle sue applicazioni.

### 7.2 Working directory

Ogni processo ha una current working directory, ovvero la directory da cui vengono interpretati tutti i percorsi relativi. Un processo può cambiare la sua directory di lavoro con la funzione chdir. Ad esempio, il percorso relativo doc/memo/joe si riferisce al file o alla directory joe, nella directory memo, nella directory doc, che deve essere una directory all'interno della working directory.

#### 7.2.1 Esempio 1.3

Per prima nell'header viene inclusa *apue.h*, un'intestazione che include alcune intestazioni di sistema standard e definisce numerose costanti e prototipi di funzioni utilizzate in tutti gli esempi

---

```

#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR          *dp;
    struct dirent  *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}

```

---

Figure 1.3 List all the files in a directory

nel testo. Successivamente, includiamo un'intestazione di sistema, dirent.h, per raccogliere i prototipi della funzione opendir e readdir, oltre alla definizione della struttura direttrice.

La funzione **opendir** restituisce un puntatore a una struttura DIR e noi passiamo questo puntatore alla funzione **readdir**. Non ci interessa cosa c'è nella struttura DIR. Quindi chiamiamo readdir in un ciclo, per leggere ogni voce della directory. La funzione readdir restituisce un puntatore a una struttura dirent o, quando ha finito con la directory, un puntatore null. Tutto ciò che esaminiamo nella struttura dirent è il nome di ciascuna voce della directory (d\_nome). Utilizzando questo nome, potremmo quindi richiamare la **funzione stat** per determinare tutti gli attributi del file. In particolare il comando stat -x è una variante del comando stat che è utile quando si vuole ottenere più dettagli sui file o directory, presentati in modo più organizzato rispetto all'output standard di stat

## 7.3 Funzione stat, fstat, fstatat e lstat

Studiamo le quattro funzioni stat utilizzate per ottenere informazioni sui file:

```

#include <sys/stat.h>

int stat(const char *restrict pathname, struct stat *restrict buf );
int fstat(int fd, struct stat *buf );
int lstat(const char *restrict pathname, struct stat *restrict buf );

```

```

int fstatat(int dirfd, const char *restrict pathname,
            struct stat *restrict buf, int flag);

```

Restituiscono 0 in caso di successo e -1 in caso di errore

Dato un *pathname*, la funzione **stat** restituisce una struttura di informazioni riguardo il file specificato. La funzione **fstat** ottiene informazioni sul file già aperto sul descrittore fd. La funzione **lstat** è simile a **stat**, ma quando il file denominato è un collegamento simbolico, tale funzione restituisce informazioni sul link simbolico, non sul file a cui fa riferimento tale link simbolico. La funzione **fstatat** fornisce un modo per restituire le statistiche del file per un percorso relativo a una directory aperta rappresentata dall'argomento fd. L'argomento flag controlla se vengono seguiti i link simbolici; quando il flag **AT\_SYMLINK\_NOFOLLOW** è impostato, **fstatat** non seguirà i link simbolici, ma piuttosto restituirà informazioni sul link stesso e non sul file a cui punta. Altrimenti, se è 0, l'impostazione predefinita è seguire i collegamenti simbolici, restituendo informazioni sul file a cui punta il collegamento simbolico. L'argomento buf è un puntatore a una struttura che dobbiamo fornire e che contiene le informazioni del file:

```

struct stat {
    mode_t      st_mode;    /* file type & mode (permissions) */
    ino_t       st_ino;     /* i-node number (serial number) */
    dev_t       st_dev;     /* device number (file system) */
    dev_t       st_rdev;    /* device number for special files */
    nlink_t     st_nlink;   /* number of links */
    uid_t       st_uid;     /* user ID of owner */
    gid_t       st_gid;     /* group ID of owner */
    off_t       st_size;    /* size in bytes, for regular files */
    struct timespec st_atim; /* time of last access */
    struct timespec st_mtim; /* time of last modification */
    struct timespec st_ctim; /* time of last file status change */
    blksize_t   st_blksize; /* best I/O block size */
    blkcnt_t    st_blocks;  /* number of disk blocks allocated */
};

```

dove il **timespec** definisce il tempo in termini di secondi e nanosecondi.

## 7.4 Tipi di file

Il tipo di file è specificato nell' **st\_mode** della struttura **stat**. Possiamo determinare il tipo di file con delle macro dove l'argomento per ciascuno di queste macro sono il membro **st\_mode** della struttura **stat**:

Macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

#### 7.4.1 Esempio 4.3

Il seguente comando stampa il tipo di file inserito come argomento da linea di comando:

---

```
#include "apue.h"
int
main(int argc, char *argv[])
{
    int      i;
    struct stat buf;
    char     *ptr;
    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            err_ret("lstat error");
            continue;
        }
        if (S_ISREG(buf.st_mode))
            ptr = "regular";
        else if (S_ISDIR(buf.st_mode))

            ptr = "directory";
        else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
        else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
        else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
        else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
        else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
        else
            ptr = "** unknown mode **";
        printf("%s\n", ptr);
    }
    exit(0);
}
```

---

In questo programma viene utilizzata la funzione *lstat* per ottenere informazioni su un file o una directory e il risultato ottenuto viene confrontato con le rispettive macro.

## 7.5 Set-User-ID e Set-Group-ID

Ogni processo ha sei o più IDs associati ad esso:

Identificatore	Descrizione
Real User ID (RUID)	Identificatore dell'utente che ha avviato il processo.
Effective User ID (EUID)	Identificatore utilizzato per determinare i permessi effettivi del processo.
Saved Set User ID (SUID)	Memorizza l'EUID originale per permettere di ritornare a esso se necessario.
Real Group ID (RGID)	Identificatore del gruppo a cui appartiene il processo.
Effective Group ID (EGID)	Identificatore che determina i permessi effettivi del processo per il gruppo.

Quando un programma ha il bit **setuid** (Set-User-ID) impostato, viene eseguito con i privilegi dell'utente proprietario del file, anziché con quelli dell'utente che lo esegue. È rappresentato con il permesso **s** nel campo dei permessi dell'utente (al posto della "x" per l'esecuzione). Il meccanismo del set-group-ID (o **setgid**, analogo al setuid) consente a un file o a un programma di essere eseguito con i permessi di gruppo del file, indipendentemente dal gruppo a cui appartiene. È rappresentato con il permesso **s** nel campo dei permessi del gruppo (al posto della "x" per l'esecuzione).

## 7.6 Funzioni access e faccessat

Le funzioni *access()* e *faccessat()* non aprono il file, ma verificano semplicemente se l'accesso è possibile controllando se un processo ha i permessi necessari per accedere a un file.

```
#include <unistd.h>

int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
```

Restituiscono 0 in caso di successo e -1 in caso di errore.

In particolare *mode* è un valore che specifica il tipo di accesso da verificare:

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission

Nell'`st_mode` sono codificati i permessi per il file, attraverso 9 bits divisi in tre categorie:

<code>st_mode</code> mask	Meaning
<code>S_IRUSR</code>	user-read
<code>S_IWUSR</code>	user-write
<code>S_IXUSR</code>	user-execute
<code>S_IRGRP</code>	group-read
<code>S_IWGRP</code>	group-write
<code>S_IXGRP</code>	group-execute
<code>S_IROTH</code>	other-read
<code>S_IWOTH</code>	other-write
<code>S_IXOTH</code>	other-execute

### 7.6.1 Esempio 4.8

Questo codice C utilizza le funzioni `access()` e `open()` per verificare i permessi di accesso a un file e poi tentare di aprirlo per la lettura.

---

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if (access(argv[1], R_OK) < 0)
        err_ret("access error for %s", argv[1]);
    else
        printf("read access OK\n");
    if (open(argv[1], O_RDONLY) < 0)
        err_ret("open error for %s", argv[1]);
    else
        printf("open for reading OK\n");
    exit(0);
}
```

---

In particolare la chiamata `access(argv[1], R_OK)` verifica se il file indicato dal percorso passato come argomento (`argv[1]`) è leggibile (controllando il permesso `R_OK`). Infine la chiamata `open(argv[1], O_RDONLY)` tenta di aprire il file in modalità di sola lettura (`O_RDONLY`).

## 7.7 Funzione umask

La funzione `umask()` (abbreviazione di "user file creation mask") è usata per impostare i permessi predefiniti che vengono bloccati quando si creano nuovi file o directory. La maschera dei permessi, o `umask`, determina quali permessi saranno disabilitati rispetto ai permessi standard

predefiniti (666 per i file, 777 per le directory) al momento della creazione di un nuovo file o directory.

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Il nuovo valore della umask, espresso in formato ottale (mode\_t).

### 7.7.1 Esempio 4.9

Il seguente programma crea due file: uno con umask azzerata e un altro con umask che disabilita i permessi di lettura e scrittura ai gruppi e agli altri.

---

```
#include "apue.h"
#include <fcntl.h>

#define RWRWRW (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)

int
main(void)
{
    umask(0);
    if (creat("foo", RWRWRW) < 0)
        err_sys("creat error for foo");
    umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
    if (creat("bar", RWRWRW) < 0)
        err_sys("creat error for bar");
    exit(0);
}
```

---

Inizialmente `#define RWRWRW` definisce una costante per i permessi che verranno assegnati ai file. A questo punto viene imposta la maschera dei permessi a 0, il che significa che non verranno bloccati permessi predefiniti al momento della creazione di un file. La chiamata `creat("foo", RWRWRW)` tenta di creare un file chiamato foo con i permessi definiti da RWRWRW (666). Il comando `umask(S_IRGRP / S_IWGRP / S_IROTH / S_IWOTH)` modifica la umask per rimuovere i permessi di lettura e scrittura per il gruppo e gli altri (022), così allo stesso modo viene creato un secondo file.

```
$ umask                                first print the current file mode creation mask
002
$ ./a.out
$ ls -l foo bar
-rw----- 1 sar                          0 Dec  7 21:20 bar
-rw-rw-rw- 1 sar                          0 Dec  7 21:20 foo
$ umask                                see if the file mode creation mask changed
002
```

## 7.8 Funzioni chmod, fchmod, fchmodat

Queste funzioni sono in grado di cambiare i permessi di un file già esistente:

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

int fchmod(int fd, mode_t mode);

int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

### 7.8.1 Esempio 4.12

Il seguente programma modifica il mode di due file.

---

```
#include "apue.h"

int
main(void)
{
    struct stat      statbuf;
    /* turn on set-group-ID and turn off group-execute */
    if (stat("foo", &statbuf) < 0)
        err_sys("stat error for foo");
    if (chmod("foo", (statbuf.st_mode & ~S_IXGRP) | S_ISGID) < 0)
        err_sys("chmod error for foo");
    /* set absolute mode to "rw-r--r--" */
    if (chmod("bar", S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH) < 0)
        err_sys("chmod error for bar");
    exit(0);
}
```

---

Dopo la memorizzazione delle informazioni del file nella struttura *statbuf*, utilizzando la chiamata *chmod* vengono modificati i suoi permessi: **statbuf.st\_mode**: Rappresenta i permessi attuali del file, **S\_IXGRP**: Inverte il bit di permesso di esecuzione per il gruppo (disabilita l'esecuzione), | **S\_ISGID**: Imposta il bit set-group-ID. Il bit set-group-ID è attivo se il file viene eseguito, e determina che i processi ereditano il gruppo del file invece del gruppo dell'utente che lo ha avviato.

```
$ ls -l foo bar
-rw-r--r-- 1 sar          0 Dec  7 21:20 bar
-rw-rwSrW- 1 sar          0 Dec  7 21:20 foo
```

## 7.9 Troncamento di un file

La funzione `truncate()` in C è utilizzata per modificare la dimensione di un file a un determinato valore. La funzione `ftruncate()` in C viene utilizzata per modificare la dimensione di un file aperto utilizzando il suo descrittore di file. In entrambi i casi se la dimensione specificata è inferiore alla dimensione attuale del file, il file verrà accorciato; se è maggiore, il file verrà esteso, e la parte aggiunta sarà riempita con zeri.

```
#include <unistd.h>

int truncate(const char *pathname, off_t length);
int ftruncate(int fd, off_t length);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

## 7.10 Funzioni `futimens`, `utimensat` e `utimes`

In UNIX e Linux, ogni file ha tre tipi principali di "timestamp" (tempi associati) che rappresentano il momento in cui è stato interagito in modi specifici:

- **Tempo di accesso (atime):** rappresenta l'ultima volta che il file è stato letto o acceso. Ad esempio, se si apre un file solo per leggerlo, questo modifica il tempo di accesso
- **Tempo di modifica (mtime):** rappresenta l'ultima volta che i dati del file sono stati modificati. Se si scrive o modifica il contenuto del file, mtime viene aggiornato
- **Tempo di cambiamento di stato (ctime):** rappresenta l'ultima volta che è stato modificato un qualsiasi attributo del file, come i permessi, il proprietario, o il contenuto stesso.

Field	Description	Example	<code>ls(1)</code> option
<code>st_atim</code>	last-access time of file data	<code>read</code>	<code>-u</code>
<code>st_mtim</code>	last-modification time of file data	<code>write</code>	<code>default</code>
<code>st_ctim</code>	last-change time of i-node status	<code>chmod, chown</code>	<code>-c</code>

Molte funzioni sono in grado di cambiare l'access time e il modification time di un file:

```

#include <sys/stat.h>

int futimens(int fd, const struct timespec times[2]);
int utimensat(int dirfd, const char *path, const struct timespec times[2],
              int flag);
int utimes(const char *pathname, const struct timeval times[2]);

```

Restituiscono 0 in caso di successo e -1 in caso di errore

Come si può notare, la funzione **futimens()** richiede come argomento il file descrittore, quindi il file deve essere necessariamente aperto. Al contrario **utimensat()** non richiede che il file sia aperto, perché utilizza un percorso di file (pathname). In entrambe le funzioni, il primo elemento dell'argomento dell'array times (times[0]) contiene l'access time (atime) e il secondo elemento (times[1]) contiene l'ora della modifica (mtime). La struttura di timespec è:

```

struct timespec {
    time_t tv_sec; // secondi
    long tv_nsec; // nanosecondi
};

```

Se times è null i timestamp saranno aggiornati al momento corrente, inoltre se uno dei campi tv\_nsec è impostato a UTIME\_NOW, quel timestamp sarà impostato al momento corrente, mentre se uno dei campi tv\_nsec è impostato a UTIME OMIT, quel timestamp non sarà modificato. Nel caso della funzione **utimes()** l'argomento times è un puntatore ad un array costituito da due timestamps (il tempo di accesso e il tempo di modifica), ma sono espressi in secondi e microsecondi:

```

struct timeval {
    time_t tv_sec; // secondi
    long tv_usec; // microsecondi
};

```

### 7.10.1 Esempio 4.21

Il seguente programma esegue la troncatura dei file mantenendo inalterati i tempi di accesso e modifica. Per fare ciò, il programma prima ottiene i tempi con la funzione *stat*, tronca il file, e poi reimposta gli orari con la funzione *futimens*.

---

```
#include "apue.h"
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int             i, fd;
    struct stat     statbuf;
    struct timespec times[2];

    for (i = 1; i < argc; i++) {
        if (stat(argv[i], &statbuf) < 0) { /* fetch current times */
            err_ret("%s: stat error", argv[i]);
            continue;
        }
        if ((fd = open(argv[i], O_RDWR | O_TRUNC)) < 0) { /* truncate */
            err_ret("%s: open error", argv[i]);
            continue;
        }
        times[0] = statbuf.st_atim;
        times[1] = statbuf.st_mtim;
        if (futimens(fd, times) < 0)           /* reset times */
            err_ret("%s: futimens error", argv[i]);
        close(fd);
    }
    exit(0);
}
```

---

Per ogni file specificato in argv, il programma esegue i seguenti passaggi: prima di tutto utilizza **stat()** per ottenere le informazioni attuali del file, come i tempi di accesso e modifica; a questo punto apre il file con i flag **O\_RDWR | O\_TRUNC**, dove: **O\_RDWR** apre il file in modalità lettura e scrittura e **O\_TRUNC** tronca il file, riducendone la lunghezza a zero byte senza eliminarlo. Dopo aver aperto il file, il programma salva i tempi originali di accesso (**st\_atim**) e modifica (**st\_mtim**) in un array di strutture **timespec**. Utilizza quindi **futimens()** per ripristinare i tempi originali del file dopo la troncatura.

## 7.11 Funzioni mkdir, mkdirat e mkdir

Le directories sono create con le funzioni **mkdir** e **mkdirat**:

```
#include <sys/stat.h>
```

```
int mkdir(const char *pathname, mode_t mode);  
int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

Restituiscono entrambe 0 in caso di successo e -1 in caso di errore

Queste funzioni creano una nuova directory vuota. Un errore comune è specificare le stesse autorizzazioni di accesso, *modes*, di un file: permessi di sola lettura e scrittura. Per una directory, infatti, normalmente vogliamo almeno uno dei bit di esecuzione abilitato, per consentire l'accesso ai nomi di file all'interno della directory.

La funzione mkdirat è simile alla funzione mkdir. Quando l'argomento fd ha il valore speciale AT\_FDCWD, o quando l'argomento pathname specifica un percorso assoluto, mkdirat si comporta esattamente come mkdir. Altrimenti, l'argomento dirfd è una directory aperta da cui verranno valutati i percorsi relativi.

Una directory aperta viene eliminata con la funzione rmdir:

```
#include <unistd.h>  
  
int rmdir(const char *pathname);
```

Restituisce 0 in caso di successo e -1 in caso di errore

Rmdir funziona solo se la directory non contiene file o altre directory. Se ci sono contenuti, bisogna prima rimuoverli, ad esempio usando funzioni come remove.

Non è possibile rimuovere la directory corrente di un processo e il processo deve avere i permessi di scrittura nella directory padre della directory da eliminare.

## 7.12 Leggere directories

Le directories possono essere lette da chiunque ha il permesso di lettura alla dierctory, e il kernel gioca un ruolo fondamentale nel controllo e nella gestione delle operazioni di scrittura e modifica delle directory, poiché il file system è una parte gestita dal kernel. Per leggere i contenuti di una directory in C, si utilizza la libreria dirent.h, che fornisce un'interfaccia per accedere alle directory e iterarne i contenuti.

Le funzioni che aprono una directory e restituiscono un puntatore a una struttura di tipo DIR:

```
#include <dirent.h>

DIR *opendir(const char *pathname);
DIR *fdopendir(int fd);
```

Restituiscono entrambe un puntatore in caso di successo e NULL in caso di errore

La funzione che legge un elemento nella directory, restituendo una struttura di tipo struct dirent:

```
#include <dirent.h>

struct dirent *readdir(DIR *dp);
```

Restituisce un puntatore in caso di successo e NULL se alla fine di una directory o errore

Una struttura di tipo struct dirent, che rappresenta un singolo elemento nella directory:

```
struct dirent {

    ino_t d_ino;           // Numero di inode
    off_t d_off;          // Offset nella directory
    unsigned short d_reclen; // Lunghezza della struttura
    unsigned char d_type; // Tipo dell'elemento (file, directory, ecc.)
    char d_name[];        // Nome dell'elemento (NULL-terminated)
};
```

La funzione che chiude una directory precedentemente aperta con opendir:

```
#include <dirent.h>

int closedir(DIR *dp);
```

Restituisce 0 in caso di successo e -1 in caso di errore

### 7.12.1 Esempio 4.22

Utilizzeremo queste routine di directory per scrivere un programma che attraversa una gerarchia di file. Tale programma accetta un singolo argomento, il percorso iniziale, e discende

ricorsivamente la gerarchia da quel punto con l'obiettivo di produrre un conteggio dei vari tipi di file. Iniziamo analizzando il main e poi tutte le altre funzioni.

```
#include "apue.h"
#include <dirent.h>
#include <limits.h>

/* function type that is called for each filename */
typedef int Myfunc(const char *, const struct stat *, int);

static Myfunc    myfunc;
static int       myftw(char *, Myfunc *);
static int       dopath(Myfunc *);

static long nreg, ndir, nblk, nchr, nfifo, nslink, nsock, ntot;

int
main(int argc, char *argv[])
{
    int      ret;

    if (argc != 2)
        err_quit("usage: ftw <starting-pathname>");

    ret = myftw(argv[1], myfunc);           /* does it all */

    ntot = nreg + ndir + nblk + nchr + nfifo + nslink + nsock;
    if (ntot == 0)
        ntot = 1;             /* avoid divide by 0; print 0 for all counts */
    printf("regular files = %ld, %.2f %%\n", nreg,
           nreg*100.0/ntot);
    printf("directories   = %ld, %.2f %%\n", ndir,
           ndir*100.0/ntot);
    printf("block special  = %ld, %.2f %%\n", nblk,
           nblk*100.0/ntot);
    printf("char special   = %ld, %.2f %%\n", nchr,
           nchr*100.0/ntot);
    printf("FIFOs          = %ld, %.2f %%\n", nfifo,
           nfifo*100.0/ntot);
    printf("symbolic links = %ld, %.2f %%\n", nslink,
           nslink*100.0/ntot);
    printf("sockets         = %ld, %.2f %%\n", nsock,
           nsock*100.0/ntot);
    exit(ret);
}
```

Inizialmente viene definito il tipo **Myfunc** che restituisce un intero e prende come argomenti un percorso (`const char*`), un puntatore ad una struttura `stat` e un intero che rappresenta il tipo di file.

Vengono definite le variabili globali per ogni tipo di file con `static long`.

La funzione **main** controlla che l'utente abbia fornito un solo argomento (il percorso di partenza), chiama `myftw`, che attraversa la directory e applica `myfunc` a ogni file e per ogni tipo di file calcola e stampa i conteggi e le percentuali rispetto al totale.

Analizziamo ora la funzione **myfunc** che viene eseguita su ogni file, quindi prepara i dati e gestisce la memoria necessaria per l'attraversamento.

```

#define FTW_F 1      /* file other than directory */
#define FTW_D 2      /* directory */
#define FTW_DNR 3    /* directory that can't be read */
#define FTW_NS 4     /* file that we can't stat */

static char *fullpath;      /* contains full pathname for every file */
static size_t pathlen;

static int             /* we return whatever func() returns */
myftw(char *pathname, Myfunc *func)
{
    fullPath = path_alloc(&pathlen);   /* malloc PATH_MAX+1 bytes */
                                         /* (Figure 2.16) */
    if (pathlen <= strlen(pathname)) {
        pathlen = strlen(pathname) * 2;
        if ((fullPath = realloc(fullPath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    strcpy(fullPath, pathname);
    return(dopath(func));
}

```

Inizialmente vengono fatte una serie di definizioni e in particolare *fullpath* è un puntatore a una stringa che contiene il percorso completo di ogni file/directory durante l'attraversamento, mentre *pathlen* è la lunghezza allocata per *fullpath*.

Nel **main** la funzione *path\_alloc* alloca dinamicamente la memoria per *fullpath*, con dimensione iniziale data da *pathlen*, e restituisce un buffer sufficientemente grande per memorizzare un percorso (tipicamente *PATH\_MAX* + 1 byte). A questo punto, come implementato nell'*if*, se il percorso iniziale (*pathname*) è più lungo di quanto attualmente allocato per *fullpath*, si rialloca memoria, infatti la dimensione del buffer viene raddoppiata rispetto alla lunghezza attuale del percorso e con la funzione *realloc* viene aumentato il buffer esistente.

A questo punto il percorso iniziale del file che stiamo considerando e passato come argomento alla funzione, *pathname*, viene copiato in *fullpath*.

Infine viene chiamata la funzione *dopath* per iniziare l'attraversamento.

Analizziamo ora la funzione **dopath** che attraversa ricorsivamente le directory e applica *myfunc* a ogni file. Inizialmente la funzione *lstat* recupera informazioni sul file specificato da *fullpath*, ricordando che in questo caso anche il link simbolico viene trattato come file. Se fallisce, chiama la funzione utente (*myftw*) con il tipo *FTW\_NS* per segnalare che il file non è stato possibile analizzare. Allo stesso modo se il file non è una directory, chiama la funzione utente con il tipo *FTW\_F* (file regolare).

A questo punto se la funzione prosegue, significa che stiamo considerando una directory, quindi

```

static int                               /* we return whatever func() returns */
dopath(Myfunc* func)
{
    struct stat      statbuf;
    struct dirent   *dirp;
    DIR             *dp;
    int              ret, n;

    if (lstat(fullpath, &statbuf) < 0) /* stat error */
        return(func(fullpath, &statbuf, FTW_NS));
    if (S_ISDIR(statbuf.st_mode) == 0) /* not a directory */
        return(func(fullpath, &statbuf, FTW_F));

    /*
     * It's a directory. First call func() for the directory,
     * then process each filename in the directory.
     */
    if ((ret = func(fullpath, &statbuf, FTW_D)) != 0)
        return(ret);

    n = strlen(fullpath);
    if (n + NAME_MAX + 2 > pathlen) { /* expand path buffer */
        pathlen *= 2;
        if ((fullpath = realloc(fullpath, pathlen)) == NULL)
            err_sys("realloc failed");
    }
    fullpath[n++] = '/';
    fullpath[n] = 0;

    if ((dp = opendir(fullpath)) == NULL) /* can't read directory */
        return(func(fullpath, &statbuf, FTW_DNR));

    while ((dirp = readdir(dp)) != NULL) {
        if (strcmp(dirp->d_name, ".") == 0 ||
            strcmp(dirp->d_name, "..") == 0)
            continue; /* ignore dot and dot-dot */
        strcpy(&fullpath[n], dirp->d_name); /* append name after "/" */
        if ((ret = dopath(func)) != 0) /* recursive */
            break; /* time to leave */
    }
    fullpath[n-1] = 0; /* erase everything from slash onward */

    if (closedir(dp) < 0)
        err_ret("can't close directory %s", fullpath);
    return(ret);
}

```

prima viene chiamata la funzione utente per la directory e poi viene preparato il buffer `pee` contenere il contenuto della directory con `n` che è pari alla lunghezza del path della directory. Anche in questo caso se `fullpath` non è abbastanza grande, raddoppia la dimensione di `fullpath` usando `realloc`. A questo punto si aggiunge prima il separatore `"/"` alla fine del percorso per costruire il prefisso dei file nella directory e poi 0 come terminatore della stringa.

Utilizzando la funzione `opendir` viene aperta la directory e se e fallisce (ad esempio, mancano i permessi per leggere la directory), chiama `func` con il tipo `FTW_DNR`. In caso contrario viene letto il contenuto della directory utilizzando un ciclo `while` con `readdir`. Ignora i nomi speciali `".."`

e ".." (rispettivamente la directory corrente e la directory padre) e costruisce il percorso completo del file o sottodirectory aggiungendo dirp->d\_name a fullname. Chiama ricorsivamente dopath per gestire il file o la sottodirectory.

A questo punto viene ripristinato il percorso fullname rimuovendo il nome del file aggiunto e viene chiusa la directory con *closedir*

Analizziamo ora la funzione **myfunc** che viene chiamata per ogni file o directory durante l'attraversamento della struttura delle directory. L'obiettivo di questa funzione è classificare i file e tenere traccia di quante volte ciascun tipo di file viene incontrato.

```
static int
myfunc(const char *pathname, const struct stat *statptr, int type)
{
    switch (type) {
    case FTW_F:
        switch (statptr->st_mode & S_IFMT) {
        case S_IFREG:   nreg++;      break;
        case S_IFBLK:   nblk++;      break;
        case S_IFCHR:   nchr++;      break;
        case S_IFIFO:   nfifo++;     break;
        case S_IFLINK:  nslink++;    break;
        case S_IFSOCK:  nsock++;     break;
        case S_IFDIR:   /* directories should have type = FTW_D */
            err_dump("for S_IFDIR for %s", pathname);
        }
        break;
    case FTW_D:
        ndir++;
        break;
    case FTW_DNR:
        err_ret("can't read directory %s", pathname);
        break;
    case FTW_NS:
        err_ret("stat error for %s", pathname);
        break;
    default:
        err_dump("unknown type %d for pathname %s", type, pathname);
    }
    return(0);
}
```

I parametri richiesti sono: il percorso completo del file o directory attualmente esaminato, puntatore a una struttura stat che contiene informazioni dettagliate sul file, un indicatore del tipo di file (FTW\_F, FTW\_D, FTW\_DNR, FTW\_NS) determinato dalla funzione che chiama myfunc.

Attraverso uno switch case vengono gestiti i quattro diversi casi che si possono incontrare effettuando un incremento per ogni tipo di file. In particolare nel caso dei file regolari, questi vengono ulteriormente classificati in base al loro tipo specifico (regolare, a blocchi, speciale a

caratteri, FIFO, link simbolico, socket) attraverso un controllo fatto usando (*statptr->st\_mode* & *S\_IFMT*), dove *S\_IFMT* è una macro di sistema che isolare i bits che rappresentano il tipo di file.

## 7.13 Funzioni chdir, fchdir e getcwd

Le funzioni chdir e fchdir cambiano la directory corrente del processo:

```
#include <unistd.h>

int chdir(const char *pathname);
int fchdir(int fd);
```

Restituiscono entrambe 0 in caso di successo e -1 in caso di errore

Mentre nella prima funzione viene specificato il percorso della nuova directory, nel secondo viene specificato un file descriptor aperto per la directory.

La funzione getcwd recupera il percorso della directory corrente del processo:

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Restituisce un puntatore al buffer in caso di successo e NULL in caso di errore

Questa funzione richiede come parametri il buffer dove viene salvato il percorso della directory corrente e la dimensione del buffer.

## 7.14 Funzioni time

Per rappresentare il tempo in termini di numero di secondi trascorsi dal 1° gennaio 1970 (00:00:00 UTC), noto come Epoch, viene utilizzato il tipo di dato **time\_t**.

La funzione **time** restituisce il valore corrente del tempo sotto forma di **time\_t**:

```
#include <time.h>

time_t time(time_t *calptr);
```

Ritorna il valore del tempo in caso di successo e  $-1$  in caso di errore

La funzione **clock\_gettime** può essere utilizzata per ottenere l'ora dell'orologio specificato e restituisce il tempo in una struttura struct timespec, che include i campi `tv_sec` (numero di secondi) e `tv_nsec` (numero di nanosecondi):

```
#include <sys/time.h>
int clock_gettime(clockid_t clock_id, struct timespec *tsp);
```

Ritorna 0 in caso di successo e  $-1$  in caso di errore

La funzione richiede come parametri un puntatore a una struttura struct timespec dove verranno memorizzati i dati del tempo e un identificatore del timer del sistema da utilizzare che può assumere i seguenti valori:

- `CLOCK_REALTIME`: Orologio di sistema (tempo corrente).
- `CLOCK_MONOTONIC`: Tempo misurato indipendentemente dal sistema operativo.
- `CLOCK_PROCESS_CPUTIME_ID`: Tempo utilizzato dal processo corrente
- `CLOCK_THREAD_CPUTIME_ID`: Tempo utilizzato dal thread corrente

La funzione **clock\_getres** restituisce la risoluzione del tempo in una struttura struct timespec, quindi indica il tempo più piccolo che può essere rappresentato dall'orologio (se la risoluzione è di 1 millisecondo, allora non sarà possibile misurare intervalli inferiori a 1 millisecondo). Questo indica il tempo minimo tra due eventi misurabili per l'orologio specificato, quindi il minimo intervallo di tempo tra due chiamate successive di `clock_gettime`:

```
#include <sys/time.h>
int clock_getres(clockid_t clock_id, struct timespec *tsp);
```

Ritorna 0 in caso di successo e  $-1$  in caso di errore

La funzione **clock\_settime** consente di aggiornare manualmente il valore di un orologio specificato, fornendo un nuovo valore in formato struct timespec:

```
#include <sys/time.h>

int clock_settime(clockid_t clock_id, const struct timespec *tsp);
```

Ritorna 0 in caso di successo e -1 in caso di errore

La funzione **gettimeofday** è utilizzata per ottenere il tempo corrente, comprensivo di entrambi: il tempo in secondi e i microsecondi:

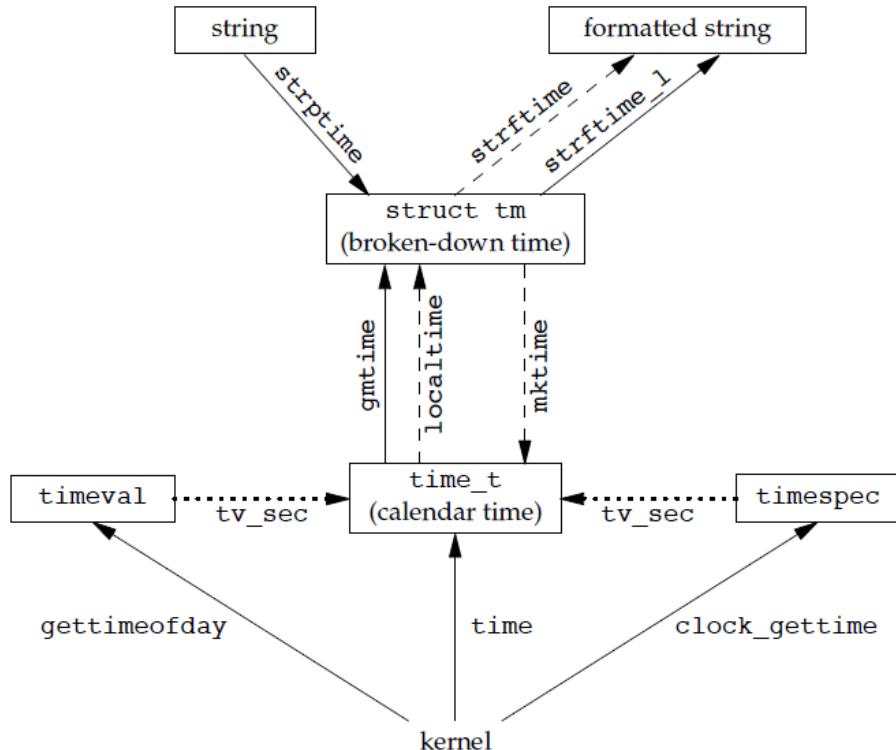
```
#include <sys/time.h>

int gettimeofday(struct timeval *restrict tp, void *restrict tzp);
```

Restituisce sempre 0

Tale funzione accetta come parametri un puntatore a una struttura `struct timeval` in cui verranno memorizzati il tempo attuale (secondi e microsecondi) e un puntatore a una struttura `struct timezone` che fornisce informazioni sulla zona oraria e può essere NULL.

Vediamo le relazioni tra le varie funzioni appena presentate:



In aggiunta vi sono le funzioni **localtime** e **gmtime** che sono utilizzate per convertire un valore

di tempo in secondi (time\_t) in una rappresentazione più leggibile come una struttura struct tm. Tuttavia, si differenziano in base al fuso orario: localtime considera il fuso orario locale, mentre gmtime considera il tempo UTC (Coordinated Universal Time):

```
#include <time.h>

struct tm *gmtime(const time_t *calptr);

struct tm *localtime(const time_t *calptr);
```

Restituiscono entrambi un puntatore ad una struttura tm o NULL in caso di errore

La **struct tm** è del tipo:

```
struct tm {

    int tm_sec;    /* secondi (0 - 59) */

    int tm_min;    /* minuti (0 - 59) */

    int tm_hour;   /* ore (0 - 23) */

    int tm_mday;   /* giorno del mese (1 - 31) */

    int tm_mon;    /* mese (0 - 11, gennaio = 0, dicembre = 11) */

    int tm_year;   /* anni dal 1900 */

    int tm_wday;   /* giorno della settimana (0 - 6, domenica = 0) */

    int tm_yday;   /* giorno dell'anno (0 - 365) */

    int tm_isdst;  /* indicazione di ora legale */

};
```

La funzione **mkttime** converte una struttura struct tm in un valore di tempo (time\_t):

```
#include <time.h>

time_t mkttime(struct tm *tmptr);
```

Restituisce un valore di tipo time\_t in caso di successo e -1 in caso di errore

Le funzioni **strftime** e **strftime\_l** sono utili quando si desidera convertire un valore di tempo (struct tm) in un formato personalizzato formattando date e orari in stringhe leggibili per l'uomo:

```
#include <time.h>

size_t strftime(char *restrict buf, size_t maxsize, const char *restrict format,
               const struct tm *restrict tmpr);

size_t strftime_l(char *restrict buf, size_t maxsize, const char *restrict format,
                  const struct tm *restrict tmpr, locale_t locale);
```

Restituiscono il numero di caratteri scritti nella stringa o 0 se buf è troppo piccolo

I parametri che le funzioni richiedono sono: un puntatore a un array di caratteri in cui viene memorizzata la stringa formattata (*buf*), la lunghezza massima della stringa (*maxsize*), la stringa di formato che specifica come deve essere formattata la data (*format*), un puntatore a una struttura struct tm contenente il tempo da formattare (*tmpr*).

La funzione **strptime** è utilizzata per convertire una stringa che rappresenta una data e un orario in una struttura struct tm:

```
#include <time.h>

char *strptime(const char *restrict buf, const char *restrict format,
               struct tm *restrict tmpr);
```

Restituisce un puntatore alla parte della stringa non convertita, oppure NULL se l'intera stringa è stata elaborata correttamente

I parametri che la funzione richiede sono: un puntatore a una stringa contenente la data e l'orario da convertire (*buf*), una stringa di formato che specifica come interpretare la stringa (*format*) e un puntatore a una struttura struct tm in cui verranno memorizzate le informazioni del tempo convertito (*tmpr*).

# Capitolo 8

## Programmi e Processi

### 8.1 Esempio 1.6

Un programma è un file eseguibile che risiede sul disco in una directory. Viene letto un programma dalla memoria e viene eseguito dal kernel come risultato di una delle sette funzioni exec. Un'istanza di esecuzione di un programma è chiamata processo. Il sistema UNIX garantisce che ogni processo abbia un identificatore numerico univoco chiamato ID process o PID.

Il seguente programma stampa i PID:

---

```
#include "apue.h"

int
main(void)
{
    printf("hello world from process ID %ld\n", (long)getpid());
    exit(0);
}
```

---

Figure 1.6 Print the process ID

Quando questo programma viene eseguito, chiama la funzione *getpid* per ottenere l'ID del processo. Questa funzione restituisce un tipo di dato pid\_t del quale non conosciamo le dimensioni e per questo viene considerato un numero intero lungo.

E' importante specificare il tipo di dato restituito poichè richiesto dalla funzione *printf* come mostrato in figura.

## 8.2 Esempio 1.7

Esistono tre funzioni principali per il controllo del processo: **fork**, **exec** e **waitpid**. (La funzione exec ha sette varianti, ma spesso ci riferiamo ad esse collettivamente semplicemente come la funzione exec)

---

```
#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    char    buf[MAXLINE];    /* from apue.h */
    pid_t   pid;
    int     status;

    printf("%% ");
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {      /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }
        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

---

Figure 1.7 Read commands from standard input and execute them

Usiamo la funzione I/O standard **fgets** per leggere una riga alla volta dal file ingresso standard. Poiché ogni riga restituita da fgets termina con un carattere di nuova riga, seguito da un byte null, utilizziamo la funzione **strlen** per calcolare la lunghezza della stringa, quindi sostituire il carattere di fine riga con un byte nullo, ovvero 0. Lo facciamo perché la funzione exclp vuole un argomento con terminazione null, non un argomento terminato a capo.

Chiamiamo **fork** per creare un nuovo processo, ovvero una copia dello spazio virtuale del processo chiamante. Quest'ultimo lo definiamo processo padre o PARENT, mentre il processo

appena creato prende il nome di CHILD ed in questo modo tale funzione ritorna due processi.  
Il fork può restituire tre tipi di valori:

- $< 0$ : in questo caso è fallito poichè l'ID del processo non può essere negativo
- $> 0$ : indica il processo parent
- $= 0$ : indica il processo figlio

Nel child viene chiamata la funzione **execlp** per azzerare lo spazio virtuale del child, il quale era copia dello spazio virtuale del parent, e quindi viene creato un nuovo spazio per eseguire un nuovo programma. La funzione execlp è una variante della famiglia exec ed in particolare il suffisso '*l*' indica che c'è una lista passata come argomento e l'altro suffisso '*p*' che viene utilizzata la variabile path.

Poiché il child chiama exclp per eseguire il nuovo programma, il genitore aspetta che il child termini. Questo viene fatto chiamando **waitpid**, specificando quale processo attendere poichè tra gli argomenti passati vi è pid, che è l'ID del processo del child. La funzione waitpid restituisce anche lo stato di terminazione del file che potrebbe essere utile per determinare come è terminato il processo figlio.

Di seguito sono state applicate alcune modifiche affinchè venissero stampati i PID dei due processi:

```
if ((pid = fork()) < 0) {
    err_sys("fork error");
} else if (pid == 0) { /* child */
    printf("Il PID del child è %ld\n", (long)getpid());
    printf("Il PID di mio padre è %ld\n", (long)getppid());
    execlp(buf, buf, (char *)0);
    err_ret("couldn't execute: %s", buf);
    exit(127);
}

/* parent */
printf("Il mio PID di genitore è %ld\n", (long)getpid());
printf("IL PID di mio figlio è %ld\n", pid);
if ((pid = waitpid(pid, &status, 0)) < 0)
    err_sys("waitpid error");
printf("%% ");
```

## 8.3 Funzione exit

La funzione più utilizzata che permette di terminare un programma normalmente:

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

Questa funzione termina l'intero processo in esecuzione e restituisce un codice di uscita al sistema operativo non continuando l'esecuzione del codice dopo la chiamata.

Il **return()** è un'istruzione del linguaggio C che se utilizzato all'interno del *main* restituisce un codice di uscita al sistema operativo e termina il programma. Se usato in altre funzioni, restituisce un valore alla funzione chiamante e continua l'esecuzione da quel punto.

### 8.3.1 Funzione atexit

La funzione atexit() in C è utilizzata per registrare funzioni che verranno eseguite automaticamente quando il programma termina normalmente, cioè quando il programma esegue la funzione exit() o return nel main().

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

Restituisce 0 in caso di successo e un numero diverso da zero in caso di errore

### 8.3.2 Esempio 7.3

Il seguente programma un esempio ben strutturato dell'uso della funzione atexit() per registrare più gestori di uscita in un programma C. Nel seguente codice la funzione atexit() registra i gestori di uscita affinchè questi vengano eseguiti quando il programma termina normalmente.

La funzione my\_exit1() viene registrata due volte e my\_exit2() una volta.

Le funzioni registrate con atexit() vengono eseguite in ordine inverso rispetto alla registrazione, come è evidente dall'output.

---

```

#include "apue.h"

static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
    if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");

    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
    if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");

    printf("main is done\n");
    return(0);
}

static void
my_exit1(void)
{
    printf("first exit handler\n");
}

static void
my_exit2(void)
{
    printf("second exit handler\n");
}

```

---

L'output atteso è:

```

main is done

first exit handler

first exit handler

second exit handler

```

## 8.4 Variabili d'ambiente

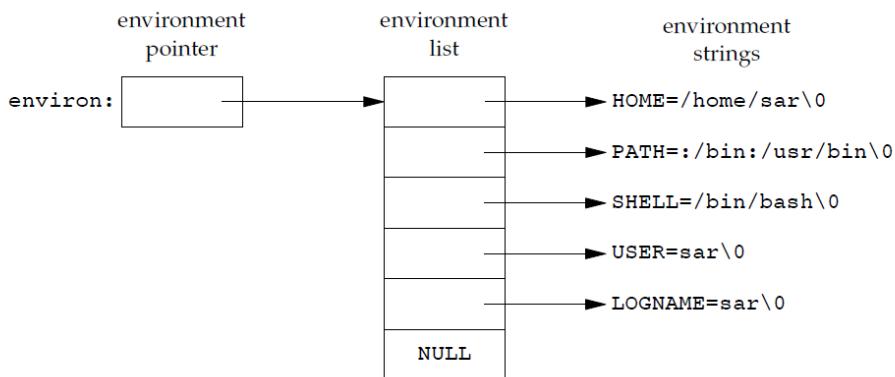
In un programma C, l'environment list (o lista delle variabili di ambiente) è un array di stringhe che rappresentano le variabili di ambiente disponibili per il processo in esecuzione.

Ogni stringa ha il formato:

NOME=VALORE

dove NOME è il nome della variabile di ambiente e VALORE è il suo contenuto.

Ad esempio se l'ambiente è costituito da cinque stringhe, risulta essere:



Nel linguaggio C, la lista delle variabili di ambiente può essere direttamente accessibile tramite la variabile globale `environ`, definita come:

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

Le variabili di ambiente possono essere lette e modificate utilizzando alcune funzioni standard di C:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Restituisce il valore della variabile di ambiente specificata da `name` oppure restituisce `NULL` se la variabile non esiste.

```
#include <stdlib.h>
int putenv(char *str);
```

Imposta una variabile di ambiente utilizzando una stringa con il formato `NOME=VALORE` e restituisce 0 in caso di successo.

#### 8.4.1 File memory\_dump.c

Inizialmente, dopo aver incluso, tutte le librerie necessarie, viene dichiarata una variabile `s` statica (quindi valida solo all'interno di questo file) inizializzata al valore 23, e la funzione `main` che viene dichiarata come `extern` poiché verrà definita più avanti nel codice.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

static const int s = 23;      /* Static var */

extern int main             /* Main appears later */
    (int argc,char *const argv[],char *const envp[]);

```

La seguente funzione **dumpAddr** stampa su un file (FILE \*p) l'indirizzo di un oggetto, insieme a una descrizione (desc). Utilizza il tipo long per convertire e stampare l'indirizzo come valore esadecimale, e il valore decimale come unsigned long. Il formato di stampa usa %08lX per mostrare l'indirizzo in formato esadecimale a 8 cifre con padding di zeri, e %lu per il valore decimale.

```

/*
 * Dump item's address to be sorted :
 */
void                         /* This must be extern */
dumpAddr(FILE *p,const char *desc,const void *addr) {

    fprintf(p,"0x%08lX %-32s\t%lu\n",
            (long)addr, desc, (unsigned long)addr);
}

```

La seguente funzione **mkPrintable** prende un buffer di caratteri (char \*buf) e sostituisce tutti i caratteri non stampabili con il simbolo '@'. Il controllo avviene usando un ciclo for che verifica se i caratteri sono fuori dal range stampabile (caratteri con valore ASCII <= 32 o >= 127).

```

/*
 * Splat out unprintable characters :
 */
static void
mkPrintable(char *buf) {

    for ( ; *buf; ++buf )
        if ( *buf <= ' ' || *buf >= 0x7F )
            *buf = '@';
}

```

La seguente funzione **fun2** è dichiarata come static, quindi visibile solo all'interno di questo file sorgente. In tale funzione viene definita una variabile locale f2 e viene usata la funzione dumpAddr per stampare il suo indirizzo insieme alla stringa "&f2".

La seguente funzione **fun1** è dichiarata come extern, quindi accessibile da altri file sorgenti che

```

/*
 * Test (static) Function 2 :
 */
static void
fun2(FILE *p) {
    int f2;

    dumpAddr(p,"&f2",&f2);
}

```

conoscono la sua dichiarazione. La funzione definisce una variabile locale f1 e stampa il suo indirizzo usando dumpAddr. Inoltre, chiama la funzione statica fun2, stmpando l'indirizzo di f2.

```

/*
 * Test (extern) Function 1 :
 */
void
fun1(FILE *p) {
    int f1;

    dumpAddr(p,"&f1",&f1);
    fun2(p);
}

```

La seguente funzione **dump** analizza e stampa informazioni come gli indirizzi di memoria delle variabili d'ambiente e altre entità (ad esempio, variabili locali, variabili statiche, funzioni standard). I risultati vengono inviati a un processo di ordinamento tramite una pipe.

```

void
dump(char * const envp[],const char *heading) {
    int x;          /* Indexes envp[x] */
    char *cp;       /* Work pointer */
    char buf[32];   /* Local buffer */
    FILE *p = 0;    /* Pipe FILE pointer */

    /*
     * Open a pipe to sort all address by
     * decreasing address, so that the last
     * displayed address will be the lowest
     * addressed item:
     */
    if ( !(p = popen("sort -n -r -k3","w")) ) {
        fprintf(stderr,"%s: %s\n",popen('sort ...');\n",
                strerror(errno));
        exit(13);
    }
    /*
     * Display a heading:
     */
    printf("\n%s\n\n",heading);
    fflush(stdout);

    /*
     * Display environment variables:
     */
    for ( x=0; envp[x] != 0; ++x ) {
        buf[0] = '&';           /* Put '&' in 1st col. */
        strncpy(buf+1,envp[x],sizeof buf-2);
        buf[sizeof buf - 1] = 0; /* Make sure we have a nul */
        mkPrintable(buf);      /* Splat out special chars */
        dumpAddr(p,buf,envp[x]);/* Dump this var's address */
    }
}

```

```

        if ( (cp = strchr(buf,'=') != 0 )
            *cp = 0;

        /*
         * Test if envp[x] is the "TERM" entry:
         */
        if ( !strcmp(buf+1,"TERM")      /* Got "TERM" ?    */
&& (cp = getenv("TERM")) != 0 /* "TERM" defined? */
&& cp != envp[x]+5 ) {      /* Ptrs mismatch? */
            strcpy(buf,"@TERM="); /* Yes, mark this. */
            strcpy(buf+6,cp,sizeof buf-7); /* Copy new val.*/
            buf[sizeof buf - 1] = 0; /* Enforce nul byte */
            mkPrintable(buf);      /* Splat out      */
            dumpAddr(p,buf,cp);   /* Dump this entry */
        }
    } /* End for () */

    dumpAddr(p,"&envp[0]",&envp[0]); /* Start of envp[] */
    sprintf(buf,"%envp[%u]",x);
    dumpAddr(p,buf,&envp[x]); /* Dump end of envp[] */

    dumpAddr(p,"&s",&s);           /* Dump static &s addr */
    dumpAddr(p,"&x",&x);           /* Dump auto &x addr */

    fun1(p);                      /* Call upon fun1()     */

    dumpAddr(p,"stderr",stderr);   /* Dump stderr address */
    dumpAddr(p,"stdout",stdout);   /* Dump stdout address */
    dumpAddr(p,"stdin",stdin);    /* Dump stdin address */

    dumpAddr(p,"fprintf",fprintf); /* Dump fprintf addr. */
    dumpAddr(p,"main",main);      /* Dump main's addr. */
    dumpAddr(p,"dump",dump);      /* Dump dump's addr. */
    dumpAddr(p,"fun1",fun1);      /* Dump fun1's addr. */
    dumpAddr(p,"fun2",fun2);      /* Dump fun2's addr. */

    if ( (cp = getenv("NVAR1")) != 0 ) /* Is NVAR1 defined? */
        dumpAddr(p,"&NVAR1",cp); /* Dump &NVAR1      */

    if ( (cp = getenv("NVAR2")) != 0 ) /* Is NVAR2 defined? */
        dumpAddr(p,"&NVAR2",cp); /* Dump &NVAR2      */

    /*
     * By closing this pipe now, we'll start sort on its
     * merry way, sorting our results. The sort output will
     * be sent to standard output.
     */
    pclose(p);
}

```

Inizialmente nell'if, viene aperta una pipe con il comando *popen* e con il comando *sort* gli indirizzi di memoria vengono ordinati in modo decrescente (-n -r).

A questo punto viene stampata l'intestazione fornita come argomento.

Il *ciclo for* analizza ciascuna variabile d'ambiente in *envp[]*. Ogni variabile inizia con & nel buffer *buf*, viene copiata con *strncpy* e resa stampabile tramite *mkPrintable*. Infine *dumpAddr* invia l'indirizzo della variabile alla pipe.

Ora viene effettuato un controllo speciale per TERM, infatti se la variabile corrente è TERM, verifica che il puntatore restituito da *getenv("TERM")* sia diverso dal puntatore di *envp[x]+5*.

Se c'è un mismatch si entra nell'if e viene stampato un messaggio con il valore aggiornato di TERM.

A questo punto il codice stampa gli indirizzi di:

- Variabili locali (&envp[0], &x) e statiche (&s)
- Puntatori di file (stderr, stdout, stdin)
- Funzioni (fprintf, main, dump, fun1, fun2)
- Variabili d'ambiente aggiuntive NVAR1 e NVAR2 se definite

Infine la chiusura della pipe avvia l'ordinamento dei dati raccolti e inviati al processo sort.

Il codice **main** implementa una serie di operazioni che manipolano le variabili d'ambiente, le analizzano e ne osservano le modifiche.

```

int
main(int argc,char * const argv[],char * const envp[]) {
    dump(envp,"INITIAL ENVIRONMENT:");

    if ( putenv("NVAR1>New Variable") == -1 ) {
        fprintf(stderr,"%s: putenv()\n", strerror(errno));
        exit(13);
    }
    if ( putenv("NVAR2>New Variable 2") == -1 ) {
        fprintf(stderr,"%s: putenv()\n", strerror(errno));
        exit(13);
    }
    sleep(5); /* per avere il tempo di invocare vmmmap */
|
    if ( putenv("TERM=oink-term") == -1 ) {
        fprintf(stderr,"%s: putenv()\n", strerror(errno));
        exit(13);
    }

/*
 * Now that we've modified our environment, let us
 * see what havoc we wreaked:
 */
dump(envp,"MODIFIED ENVIRONMENT:");
sleep(1); /* This waits for the sort output */

/*
 * This simple test just proves that the new value
 * for TERM has been changed.
 */
printf("\ngetenv('TERM') = '%s';\n",getenv("TERM"));
return 0;
}

```

La funzione *dump* viene chiamata per esaminare e stampare l'ambiente iniziale (contenuto di *envp[]*). Questo include tutte le variabili d'ambiente fornite al programma al momento della sua esecuzione.

Vengono create due nuove variabili d'ambiente con la funzione *puntev*, in particolare NVAR1 è impostata a "New Variable" e NVAR2 è impostata a "New Variable 2".

La variabile d'ambiente TERM viene modificata, assegnandole il valore "oink-term" e questo consente di verificare come un cambio a una variabile d'ambiente preesistente influenzi il programma.

Viene chiamata nuovamente la funzione *dump* per stampare l'ambiente aggiornato e si usa *getenv* per ottenere e stampare il valore attuale di TERM dopo la modifica per verificare che il valore sia stato correttamente aggiornato a "oink-term".

## 8.5 Allocazione di memoria

L'allocazione di memoria è una parte fondamentale della programmazione in C (e in altri linguaggi), che consente di gestire la memoria dinamicamente durante l'esecuzione di un programma.

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);
```

Restituiscono un puntatore non nullo in caso di successo e NULL in caso di errore

Nello specifico:

- **malloc()**: Alloca un blocco di memoria non inizializzata di una dimensione specificata.
- **calloc()**: Alloca un blocco di memoria inizializzata a zero, utile per evitare di dover azzerare manualmente i valori.
- **realloc()**: Ridimensiona un blocco di memoria precedentemente allocato.

## 8.6 Funzioni getrlimit e setrlimit

Le funzioni *getrlimit* e *setrlimit* in C sono utilizzate per ottenere e modificare i limiti delle risorse di sistema (come la memoria, il numero di file aperti, il tempo di CPU, ecc.) a livello di processo.

```
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlptr);

int setrlimit(int resource, const struct rlimit *rlptr);
```

Restituicono 0 in caso di successo e -1 in caso di errore

Entrambe le funzioni specificano una sola *resource*, ovvero la risorsa di sistema di cui si vuole ottenere il limite, e un puntatore alla seguente struttura:

```
struct rlimit {  
    rlim_t rlim_cur; // Limite corrente  
    rlim_t rlim_max; // Limite massimo  
};
```

### 8.6.1 Esempio 7.16

Il programma mostra come utilizzare la funzione getrlimit per ottenere i limiti delle risorse di sistema di un processo. Viene stampato un elenco di limiti, insieme ai limiti correnti e massimi di ciascuna risorsa. Nel main viene eseguita una serie di chiamate a doit, che stampa i limiti per le risorse del sistema, come RLIMIT\_CPU, RLIMIT\_STACK, RLIMIT\_FSIZE, ecc. Di seguito la funzione pr\_limits() che riceve il nome della risorsa e il valore della costante della risorsa:

```
static void  
pr_limits(char *name, int resource)  
{  
    struct rlimit      limit;  
    unsigned long long lim;  
  
    if (getrlimit(resource, &limit) < 0)  
        err_sys("getrlimit error for %s", name);  
    printf("%-14s ", name);  
    if (limit.rlim_cur == RLIM_INFINITY) {  
        printf("(infinite) ");  
    } else {  
        lim = limit.rlim_cur;  
        printf("%10lld ", lim);  
    }  
    if (limit.rlim_max == RLIM_INFINITY) {  
        printf("(infinite)");  
    } else {  
        lim = limit.rlim_max;  
        printf("%10lld", lim);  
    }  
    putchar((int)'\n');  
}
```

Usa la funzione getrlimit per ottenere i limiti di quella risorsa, quindi stampa prima il nome della

risorsa, se il limite corrente (rlim\_cur) o massimo (rlim\_max) è infinito (RLIM\_INFINITY), viene stampato "(infinite)" e se il limite è numerico, viene stampato come un numero a 64 bit (long long).

## 8.7 Funzione fork

La funzione fork serve per creare un nuovo processo (chiamato figlio) che è una copia del processo chiamante (chiamato padre).

```
#include <unistd.h>
pid_t fork(void);
```

Restituisce 0 se siamo nel processo figlio, un valore maggiore di 0 se siamo nel processo padre (PID del padre) e -1 in caso di errore

Quando un processo viene duplicato tramite fork, la memoria del padre e del figlio viene inizialmente condivisa. La memoria non viene effettivamente copiata fino a quando uno dei due processi non cerca di modificarla. Questa tecnica ottimizza l'uso della memoria, poiché evita la copia inutile dei dati fino a quando non è necessario. Dopo il fork, il processo padre e il processo figlio sono due entità indipendenti.

Quando un processo padre crea un file o apre un file con una funzione come open(), il sistema operativo associa un descrittore di file al processo. Questo descrittore viene copiato nel processo figlio dopo un fork(). Tuttavia, il file non viene duplicato; entrambi i processi (padre e figlio) condivideranno lo stesso descrittore di file. Quando un file è aperto, il sistema tiene traccia della posizione del puntatore del file (ossia la posizione in cui verrà letta o scritta la prossima informazione). Dopo il fork(), padre e figlio condividono la stessa posizione del puntatore nel file. Ciò significa che, se uno dei processi modifica la posizione del puntatore (ad esempio, tramite lseek() o read()/write()), l'altro processo vedrà quel cambiamento, poiché il puntatore è condiviso.

## 8.8 Funzioni wait e waitpid

Le funzioni wait e waitpid in C vengono utilizzate per la gestione dei processi figli. Entrambe permettono a un processo padre di aspettare la terminazione di uno o più processi figli. Sono essenziali per evitare che i processi figli diventino processi orfani (ossia processi che non hanno un processo padre), gestendo correttamente il loro stato di uscita e le risorse.

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Restituiscono il PID del processo figlio che è terminato o  $-1$  in caso di errore

La funzione wait permette al processo padre di aspettare la terminazione di uno qualsiasi dei suoi figli. Se il processo padre ha più figli, wait restituirà il PID del primo figlio che termina. Se ci sono più figli che terminano nello stesso momento, non c'è garanzia su quale verrà selezionato per primo.

La funzione waitpid è più potente di wait poiché offre il controllo su quale processo il padre sta aspettando, e consente di fare altre operazioni (come non bloccare il processo padre) mentre si aspetta la terminazione di un processo figlio.

Entrambe le funzioni wait e waitpid restituiscono lo stato di uscita del processo figlio attraverso il parametro statloc.

- WIFEXITED(status): Verifica se il processo figlio è terminato normalmente (cioè con `exit()` o `return()`).
- WIFSIGNALED(status): Verifica se il processo figlio è terminato a causa di un segnale non gestito.
- WIFSTOPPED(status): Verifica se il processo figlio è stato sospeso da un segnale (ad esempio, con `kill`).
- WIFCONTINUED(status): Verifica se il processo figlio è stato ripreso dopo una sospensione.

### 8.8.1 Esempio 8.5

Il codice in figura è una funzione pr\_exit() in C che stampa informazioni sullo stato di terminazione di un processo figlio. La funzione prende come parametro status, che è lo stato di uscita di un processo, tipicamente restituito da wait() o waitpid(). In questa funzione, vengono utilizzate diverse macro per determinare come il processo figlio è terminato e per stampare un messaggio appropriato.

---

```
#include "apue.h"
#include <sys/wait.h>
void
pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
               WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
               WTERMSIG(status),
#ifndef WCOREDUMP
               WCOREDUMP(status) ? " (core file generated)" : "");
#else
               "");
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
               WSTOPSIG(status));
}
```

---

Questa funzione verrà chiamata nel programma dell'esempio 8.6, ottenendo come output su Linux:

```
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

### 8.8.2 Esempio 8.6

Il codice mostra un programma C che crea tre processi figli tramite la funzione fork() e per ciascuno di questi figli, esegue operazioni che generano diversi tipi di terminazione (normale, causata da un segnale, e per errore aritmetico).

Alla fine, il processo padre attende ogni figlio e stampa lo stato di terminazione di ciascun processo utilizzando la funzione pr\_exit().

---

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;
    int      status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        exit(7);

    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        abort();                 /* generates SIGABRT */
    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)           /* child */
        status /= 0;              /* divide by 0 generates SIGFPE */
    if (wait(&status) != pid)    /* wait for child */
        err_sys("wait error");
    pr_exit(status);            /* and print its status */

    exit(0);
}

```

---

### 8.8.3 Esempio 8.8

Il programma è un esempio che esplora il comportamento del processo padre e dei figli durante l'uso di `fork()` e `waitpid()`. Il programma crea due processi figli in sequenza e utilizza `sleep()` per simulare l'esecuzione dei processi figli, con un focus particolare su come il sistema gestisce i genitori di processo e il riordino dei processi (reaping) tramite il processo `init`.

**Fork del primo processo figlio:** Il primo `fork()` crea il primo processo figlio. Questo processo figlio crea un secondo processo figlio tramite un altro `fork()`.

**Nel primo figlio:** Quando il primo processo figlio esegue il secondo `fork()`, il processo figlio originale diventa il "padre" del secondo processo figlio. Subito dopo, il primo figlio termina con `exit(0)`, lasciando il secondo figlio come orfano. A questo punto, il sistema operativo assegnerà il processo `init` (PID 1) come nuovo "genitore" del secondo figlio (orfano). Quando il secondo figlio termina, sarà `init` a raccogliere lo stato del suo processo, non il primo figlio, che è già terminato.

**Nel secondo figlio:** Il secondo processo figlio, creato dal primo figlio, esegue sleep(2) per due secondi, il che lo rende orfano (perché il primo figlio termina prima). Durante questa pausa, il suo genitore diventa init, il quale successivamente raccoglierà lo stato di terminazione del secondo figlio. Il secondo figlio stampa il proprio PPID, che sarà init (PID 1), poiché il primo figlio è terminato.

**Nel padre:** Il processo padre originale attende la terminazione del primo figlio tramite waitpid(). Dopo aver atteso il primo figlio, il padre termina.

---

```

#include "apue.h"
#include <sys/wait.h>

int
main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* first child */
        if ((pid = fork()) < 0)
            err_sys("fork error");
        else if (pid > 0)
            exit(0); /* parent from second fork == first child */

        /*
         * We're the second child; our parent becomes init as soon
         * as our real parent calls exit() in the statement above.
         * Here's where we'd continue executing, knowing that when
         * we're done, init will reap our status.
         */
        sleep(2);
        printf("second child, parent pid = %ld\n", (long)getppid());
        exit(0);
    }
    if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}

```

---

## 8.9 Race Conditions

Le race conditions (condizioni di gara) si verificano in un programma quando il comportamento del sistema dipende dall'ordine in cui vengono eseguiti gli eventi concorrenti, come i thread o i processi. In altre parole, si verifica una "race" (corsa) tra due o più operazioni concorrenti che tentano di accedere e manipolare una risorsa condivisa. Se queste operazioni non sono

correttamente sincronizzate, il risultato finale può dipendere dall'ordine di esecuzione, portando a comportamenti imprevedibili e bug difficili da individuare.

### 8.9.1 Esempio 8.12

Il programma esplora il comportamento dei processi creati con fork() e come viene gestita la scrittura su stdout da parte di processi concorrenti. In particolare, il programma dimostra un esempio di output concorrente in cui sia il processo padre che il processo figlio scrivono sulla stessa risorsa di output standard (stdout).

Poiché i processi padre e figlio stanno cercando di scrivere su stdout senza sincronizzazione, l'output potrebbe apparire interlacciato, ad esempio:

output from parent	output from child	ououtput from parent
output from child	output from parent	from childt from parent

---

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t    pid;
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;
    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

---

### 8.9.2 Esempio 8.13

Il seguente codice è stato fatto sulla base dell'esempio precedente, il quale è stato modificato aggiungendo le funzioni **WAIT** e **TELL** per evitare la race condition.

---

```

#include "apue.h"
static void charatatime(char *);
int
main(void)
{
    pid_t pid;
+    TELL_WAIT();
+
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
+        WAIT_PARENT(); /* parent goes first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
+        TELL_CHILD(pid);
    }
    exit(0);
}
static void
charatatime(char *str)
{
    char *ptr;
    int c;
    setbuf(stdout, NULL); /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}

```

---

Come possiamo notare utilizzando la funzione **WAIT\_PARENT()** il processo figlio attende che il processo padre completi il suo output prima di iniziare, mentre il processo padre, una volta completato, segnala al processo figlio di procedere utilizzando **TELL\_CHILD**.

Poichè ora i due processi sono stati sincronizzati, l'output sarà:

```

output from parent
output from child

```

Le funzioni **TELL\_CHILD()** e **WAIT\_PARENT()** non sono funzioni o macro della libreria C quindi è probabilmente definito in qualche file incluso o parte di una libreria personalizzata, ovvero apue.h che è incluso nel codice. Per trovare la definizione basta eseguire questo comando:

```
grep -r 'TELL_CHILD' /path/to/include/
```

In questo modo nel file *apue.h* si trova:

```

void      TELL_WAIT(void);
void      TELL_PARENT(pid_t);           void      TELL_CHILD(pid_t);
void      WAIT_PARENT(void);          void      WAIT_CHILD(void);

```

## 8.10 Funzioni exec

Le funzioni della famiglia exec in C servono per eseguire un nuovo programma all'interno di un processo esistente. Quando una funzione exec viene chiamata, il programma corrente viene sostituito dal nuovo programma specificato. L'area di memoria del processo viene rimpiazzata e l'esecuzione riparte dal nuovo programma e l'identificatore del processo (PID) rimane invariato. La famiglia di funzioni exec include diverse varianti, che si differenziano per il modo in cui accettano gli argomenti e specificano il file eseguibile:

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

Tutte restituiscono  $-1$  in caso di errore e non ritornano in caso di successo

Gli argomenti per queste sette funzioni exec sono difficili da ricordare. Le lettere nei nomi delle funzioni aiutano in qualche modo.

Function	pathname	filename	fd	Arg list	argv[ ]	environ	envp[ ]
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•		•		•		•
fexecve					•		•
(letter in name)		p	f	l	v		e

La *lettera p* significa che la funzione accetta come argomento il nome del file e utilizza la variabile d'ambiente PATH per trovare il file eseguibile. La *lettera l* significa che la funzione accetta un elenco di argomenti ed è mutuamente esclusiva con la *lettera v*, il che significa che

accetta un vettore argv[]]. Infine, la *lettera e* significa che la funzione accetta un array envp[] invece di utilizzare l'ambiente corrente.

### 8.10.1 Esempio 8.16

Questo programma dimostra l'uso delle funzioni della famiglia exec per eseguire un nuovo programma in due scenari distinti:

- **execle:** Specifica il percorso completo del programma da eseguire e un ambiente personalizzato.
- **execlp:** Specifica solo il nome del programma e utilizza la variabile PATH per individuarlo, ereditando l'ambiente corrente.

---

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t    pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                   "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) { /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

---

Inizialmente viene inizializzato un array di stringhe che rappresenta **l'ambiente personalizzato** che sarà passato al programma eseguito da execle e contiene due variabili: USER e PATH. A questo punto nel main viene effettuato il *primo Fork* che crea un processo figlio e con la funzione **execle** viene eseguito il programma di cui è stato specificato il path e con l'ambiente

personalizzato precedentemente.

Il processo padre aspetta la terminazione del primo processo figlio e poi viene effettuata un **secondo Fork** che crea un altro processo figlio. A questo punto con la funzione **execvp** il sistema crea il programma (del quale è stato specificato solo il nome) nella variabile PATH ereditata dall'ambiente corrente.

L'output iniziale era:

```
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
execvp error: No such file or directory
```

Questo accade perchè execvp cerca il programma solo nelle directory elencate nella variabile d'ambiente PATH. Se echoall non è in una di queste directory, il sistema non riesce a trovarlo. Per risolvere questo problema è possibile aggiungere temporaneamente la cwd al PATH:

```
PATH = ${PATH}:.
```

In questo modo anche la funzione execvp con l'ambiente corrente.

## 8.11 Funzioni setuid e setgid

Le funzioni setuid e setgid sono meccanismi utilizzati per gestire i permessi di accesso ai file e ai processi, permettendo di eseguire operazioni con i privilegi di un determinato utente o gruppo. La funzione setuid consente di cambiare l'UID effettivo del processo chiamante. In termini semplici, permette a un processo di eseguire operazioni come se fosse l'utente specificato. La funzione setgid funziona in modo analogo a setuid, ma si applica al Group ID (GID) invece che all'UID:

```
#include <unistd.h>
int setuid(uid_t uid);
```

```
int setgid(gid_t gid);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

## 8.12 Funzioni setreuid e setregid

Le funzioni setreuid e setregid sono estensioni delle funzioni setuid e setgid consentendo di cambiare il primo sia l'UID effettivo che quello reale di un processo e il secondo sia l'GID effettivo che quello reale di un processo:

```
#include <unistd.h>  
  
int setreuid(uid_t ruid, uid_t euid);  
  
int setregid(gid_t rgid, gid_t egid);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

## 8.13 Funzioni seteuid e setegid

Le funzioni seteuid e setegid consentono di modificare temporaneamente solo l'UID o GID effettivo di un processo, senza modificare l'UID o GID reale:

```
#include <unistd.h>  
  
int seteuid(uid_t uid);  
  
int setegid(gid_t gid);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

## 8.14 Funzione system

La funzione system() è una funzione utilizzata per eseguire comandi o script esterni al programma corrente:

```
#include <stdlib.h>  
  
int system(const char *cmdstring);
```

Restituisce 0 in caso di successo e  $-1$  in caso di errore

Tale funzione ammette come parametro una stringa che rappresenta il comando o script da eseguire (*cmdstring*).

## 8.15 Process Times

Precedentemente è stato visto che si possono misurare tre diversi tempi: tempo dell'orologio, tempo utente e tempo di sistema. Nel caso di un processo si possono ottenere questi valori usando la funzione `times`:

```
#include <sys/times.h>
clock_t times(struct tms *buf );
```

Restituisce i tempi totali impiegati dal processo per eseguire operazioni sia in modalità utente (user time) che in modalità sistema (system time), oppure  $-1$  in caso di errore

Questa funzione inserisce i tempi nella **struct tms** alla quale punta *buf*:

```
struct tms {
    clock_t tms_utime; /* user CPU time */
    clock_t tms_stime; /* system CPU time */
    clock_t tms_cutime; /* user CPU time, terminated children */
    clock_t tms_cstime; /* system CPU time, terminated children */
};
```

Tale funzione Restituisce un valore di tipo `clock_t` che rappresenta il numero di ticks trascorsi, ovvero l'unità temporale del sistema operativo. Il valore restituito, quindi, potrebbe avere molta importanza se utilizzato per fare la differenza con il valore restituito da un altro processo.

Esempio di output:

```
Tempo utente: 12345 ticks
Tempo sistema: 6789 ticks
Tempo dei figli utente: 5432 ticks
Tempo dei figli sistema: 1234 ticks
```

### 8.15.1 Esempio 8.31

Il programma seguente esegue ogni argomento della riga di comando come una stringa di comando della shell, temporizzando il comando e stampando i valori dalla struttura tms.

```
#include "apue.h"
#include <sys/types.h>

static void pr_times(clock_t, struct tms *, struct tms *);
static void do_cmd(char *);

int
main(int argc, char *argv[])
{
    int      i;

    setbuf(stdout, NULL);
    for (i = 1; i < argc; i++)
        do_cmd(argv[i]); /* once for each command-line arg */
    exit(0);
}

static void
do_cmd(char *cmd)      /* execute and time the "cmd" */
{
    struct tms  tmsstart, tmssend;
    clock_t    start, end;
    int       status;

    printf("\ncommand: %s\n", cmd);

    if ((start = times(&tmsstart)) == -1) /* starting values */
        err_sys("times error");

    if ((status = system(cmd)) < 0)           /* execute command */
        err_sys("system() error");

    if ((end = times(&tmssend)) == -1)        /* ending values */
        err_sys("times error");

    pr_times(end-start, &tmsstart, &tmssend);
    pr_exit(status);
}

static void
pr_times(clock_t real, struct tms *tmsstart, struct tms *tmssend)
{
    static long      clkck = 0;

    if (clkck == 0) /* fetch clock ticks per second first time */
        if ((clkck = sysconf(_SC_CLK_TCK)) < 0)
            err_sys("sysconf error");

    printf("  real:  %f\n", real / (double) clkck);
    printf("  user:  %f\n",
           (tmssend->tms_utime - tmsstart->tms_utime) / (double) clkck);
    printf("  sys:   %f\n",
           (tmssend->tms_stime - tmsstart->tms_stime) / (double) clkck);
    printf("  child user: %f\n",
           (tmssend->tms_cutime - tmsstart->tms_cutime) / (double) clkck);
    printf("  child sys:  %f\n",
           (tmssend->tms_cstime - tmsstart->tms_cstime) / (double) clkck);
}
```

Inizialmente vengono dichiarate due funzioni: la funzione *pr\_times()* stampa i tempi di CPU utilizzati dal processo, mentre la funzione *do\_cmd()* segue un comando e misura il tempo im-

piegato per l'esecuzione.

La funzione **main** prende in input una serie di comandi come argomenti da riga di comando e li esegue uno alla volta utilizzando `do_cmd(argv[i])` e misura il tempo di esecuzione.

La funzione **do\_cmd** inizialmente utilizzando `times(&tmsstart)` misura il tempo di avvio, quindi prima dell'esecuzione del programma; poi utilizza la funzione `system()` per eseguire il comando specificato e infine Misura il tempo di fine esecuzione utilizzando `times(&tmsend)`. A questo punto chiama la funzione `pr_times` per visualizzare i tempi di CPU utilizzati.

La funzione **pr\_times** stampa i tempi di CPU utilizzati dal processo.

Se eseguiamo il programma, un esempio di output sarà:

```
$ ./a.out "sleep 5" "man bash >/dev/null"
```

```
command: sleep 5
real: 5.01
user: 0.00
sys: 0.00
child user: 0.00
child sys: 0.00
normal termination, exit status = 0
```

```
command: man bash >/dev/null
real: 1.46
user: 0.00
sys: 0.00
child user: 1.32
child sys: 0.07
normal termination, exit status = 0
```

# Capitolo 9

## Segnali

I segnali, chiamati anche software interrupts, sono degli impulsi asincroni trasmessi da un processo ad un altro permettendo quindi la comunicazione tra questi, e sono una tecnica utilizzata per notificare a un processo che si è verificata una condizione. I segnali sono definiti impulsi asincroni perché possono arrivare in un processo in qualsiasi momento, indipendentemente da ciò che il processo sta facendo. Per esempio, se un processo divide per zero, il segnale il cui nome è **SIGFPE** (floating-point eccezione) viene inviato al processo. Il processo prevede tre scelte per gestire il file segnale:

- *Ignora il segnale* Questa opzione non è consigliata per i segnali che denotano a eccezione hardware, come la divisione per zero o il riferimento alla memoria esterna lo spazio degli indirizzi del processo, poiché i risultati non sono definiti
- *Lascia che si verifichi l'azione predefinita* In questo caso viene fatta eseguire l'azione di default che ad esempio per una condizione di divisione per zero, l'impostazione predefinita è di terminare il processo. E' possibile consultare queste azioni di default tramite il comando **man signal**
- *Gestire il segnale* Viene fornita una funzione che viene chiamata quando si verifica il segnale, in questo modo sapremo quando il segnale si verifica e possiamo gestirlo come desideriamo

## 9.1 Esempio 1.10

Per analizzare il seguente programma, facciamo riferimento all'esempio 1.7 (8.2), al quale è stato aggiunto un esempio per spiegare la gestione dei segnali:

```
#include "apue.h"
#include <sys/wait.h>

+ static void sig_int(int);           /* our signal-catching function */
+
+ int
main(void)
{
    char     buf[MAXLINE];    /* from apue.h */
    pid_t    pid;
    int      status;

+    if (signal(SIGINT, sig_int) == SIG_ERR)
+        err_sys("signal error");
+
    printf("%% "); /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        if (buf[strlen(buf) - 1] == '\n')
            buf[strlen(buf) - 1] = 0; /* replace newline with null */

        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) { /* child */
            execlp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ((pid = waitpid(pid, &status, 0)) < 0)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
+
+ void
+ sig_int(int signo)
+ {
+     printf("interrupt\n%% ");
+ }
```

Figure 1.10 Read commands from standard input and execute them

Se invochiamo questo programma e premiamo il tasto di interruzione, il processo termina perché l'azione predefinita per questo segnale, denominato SIGINT, consiste nel terminare il processo. Il processo non ha detto al kernel di fare qualsiasi cosa diversa da quella predefinita con questo segnale, quindi il processo termina. Per gestire questo segnale, il programma deve chiamare la funzione signal, specificando il nome della funzione da chiamare quando viene generato il segnale SIGINT. La funzione è denominata sig\_int; quando viene chiamato, stampa semplicemente un messaggio e un nuovo prompt. Possiamo trovare la definizione di err\_sys in lib/error.c.

## 9.2 Funzione signal

La funzione signal permette di definire come un processo debba comportarsi quando riceve un determinato segnale, quindi serve ad installare un gestore.

```
#include <signal.h>

void (*signal(int signo, void (*func)(int)))(int);
```

Restituisce il gestore precedente del segnale (cioè il valore di func prima della chiamata) in caso di successo, oppure restituisce SIG\_ERR in caso di errore

L'argomento *signo* è semplicemente il nome del segnale, mentre il valore di *func* è (a) la costante SIG\_IGN, (b) la costante SIG\_DFL o (c) l'indirizzo di una funzione da chiamare quando si verifica il segnale. Se specifichiamo SIG\_IGN, stiamo dicendo al sistema di ignorare il segnale. (Ricordiamo che non possiamo ignorare i due segnali SIGKILL e SIGSTOP.) Quando specifichiamo SIG\_DFL, viene ripristinato il comportamento predefinito del segnale.

### 9.2.1 Esempio 10.2

Il seguente programma dimostra come gestire i segnali personalizzati SIGUSR1 e SIGUSR2 utilizzando la funzione signal.

---

```
#include "apue.h"

static void sig_usr(int); /* one handler for both signals */

int
main(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("can't catch SIGUSR2");
    for ( ; ; )
        pause();
}

static void
sig_usr(int signo) /* argument is signal number */
{
    if (signo == SIGUSR1)
        printf("received SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("received SIGUSR2\n");
    else
        err_dump("received signal %d\n", signo);
}
```

---

Le chiamate a `signal(SIGUSR1, sig_usr)` e `signal(SIGUSR2, sig_usr)` associano il gestore `sig_usr` ai segnali SIGUSR1 e SIGUSR2. In caso di errore durante l'associazione, il programma chiama `err_sys` per stampare un messaggio di errore e terminare.

La funzione `sig_usr` è il gestore dei segnali e prende un parametro `signo` che è il numero del segnale ricevuto. Quando il processo riceve un segnale:

- Se il segnale è SIGUSR1, stampa received SIGUSR1.
- Se il segnale è SIGUSR2, stampa received SIGUSR2.
- Per qualsiasi altro segnale, chiama `err_dump`, probabilmente per registrare un errore e terminare.

Il programma entra in un ciclo infinito (`for (;;)`) e chiama `pause()`. La funzione `pause()` mette il processo in uno stato di attesa finché non arriva un segnale. Quando un segnale viene ricevuto, il gestore associato è eseguito, e successivamente il programma torna in attesa con `pause()`.

Invochiamo il programma in background e utilizziamo il comando `kill(1)` per inviargli segnali. Quando inviamo il segnale SIGTERM, il processo viene terminato, poiché non cattura il segnale e l'azione predefinita per il segnale è la terminazione.

```
$ ./a.out &                                start process in background
[1]    7216                               job-control shell prints job number and process ID
$ kill -USR1 7216                           send it SIGUSR1
received SIGUSR1
$ kill -USR2 7216                           send it SIGUSR2
received SIGUSR2
$ kill 7216                                now send it SIGTERM
[1]+  Terminated                  ./a.out
```

## 9.3 Interrupted System Calls

Una caratteristica dei precedenti sistemi UNIX era che se un processo era bloccato in una chiamata di sistema "lenta" (ad esempio, `read`, `write`, `accept`, `select`, ecc.), l'arrivo di un segnale interrompeva la chiamata. La chiamata di sistema terminava immediatamente, restituendo un errore, e `errno` veniva impostato su `EINTR` (interrupted system call).

Nei sistemi UNIX e POSIX più recenti, è stata introdotta un'opzione per gestire automaticamente il ripristino delle chiamate di sistema dopo un'interruzione da segnale: con la funzione `sigaction`, è possibile specificare il flag `SA_RESTART` durante l'installazione di un gestore di segnali. Se questo flag è attivo, le chiamate di sistema interrotte da un segnale vengono automaticamente riprese dal kernel, senza che il programmatore debba esplicitamente gestire `EINTR`.

## 9.4 Funzioni Reentrant

Quando un segnale che viene catturato viene gestito da un processo, la normale sequenza di istruzioni eseguite dal processo viene temporaneamente interrotta dal segnale gestore. Quando il gestore termina (ad esempio, con `return`), il kernel ripristina lo stato salvato del processo. I gestori di segnali sono eseguiti in contesti asincroni, anche mentre il processo sta utilizzando risorse condivise. Pertanto, devono usare solo funzioni reentrant (sicure per i segnali). Alcune funzioni della libreria standard non sono signal-safe, ovvero non possono essere chiamate in sicurezza dai gestori di segnali, perché potrebbero usare risorse condivise (ad esempio, buffer interni, heap, file descriptor). Funzioni come `printf`, `malloc`, `free`, e molte altre non sono reentrant e quindi non devono essere usate nei gestori di segnali.

## 9.5 Reliable-Signal Terminology and Semantics

Diciamo che un segnale viene inviato (*delivered*) a un processo quando viene intrapresa l'azione relativa a un segnale. Durante il tempo che intercorre tra la generazione di un segnale e la sua consegna, si dice che il segnale sia essere in sospeso (*pending*). Un processo ha la possibilità di bloccare (*blocking*) la consegna di un segnale. Se un segnale bloccato viene generato per un processo e se l'azione per quel segnale è l'azione predefinita o per catturare il segnale, allora il segnale rimane in sospeso per il processo finché il processo (a) sblocca il segnale o (b) cambia l'azione per ignorare il segnale. Il sistema determina cosa fare con un segnale bloccato quando il segnale viene inviato, non quando viene generato. Ciò consente al processo di modificare l'azione per il segnale prima che venga consegnato.

## 9.6 Funzioni kill e raise

La funzione `kill` invia un segnale a un processo o a un gruppo di processi. La funzione `raise` consente a un processo di inviare un segnale a se stesso.

```
#include <signal.h>

int kill(pid_t pid, int signo);

int raise(int signo);
```

Restituiscono 0 in caso di successo e -1 in caso di errore

Nella funzione `kill` il parametro *pid* specifica il destinatario del segnale:

- $\text{pid} > 0$ : Invia il segnale al processo con identificatore PID specificato.
- $\text{pid} == 0$ : Invia il segnale a tutti i processi nel gruppo di processi del chiamante.
- $\text{pid} < -1$ : Invia il segnale a tutti i processi nel gruppo di processi con identificatore -*pid*.
- $\text{pid} == -1$ : Invia il segnale a tutti i processi che il chiamante ha il permesso di segnalare

Mentre il parametro *signo* specifica il segnale da inviare.

## 9.7 Funzioni alarm e pause

La funzione `alarm` è utilizzata per inviare un segnale a un processo dopo un determinato intervallo di tempo, generalmente per implementare un timeout o una scadenza per un'operazione.

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

La funzione restituisce il valore del timer precedentemente impostato (ovvero, il numero di secondi rimasti prima del prossimo segnale SIGALRM), oppure 0 se non c'era un timer attivo. Il parametro *seconds* specifica numero di secondi dopo i quali il processo riceverà il segnale SIGALRM.

La funzione `pause` è una funzione che sospende l'esecuzione del processo chiamante fino a quando non viene ricevuto un segnale che il processo può gestire.

```
#include <unistd.h>
int pause(void);
```

La funzione pause non restituisce mai normalmente (a meno che non si verifichi un errore). Se la chiamata fallisce, restituisce -1 e imposta errno per indicare l'errore.

### 9.7.1 Esempio

Questo programma in C esegue un ciclo infinito di moltiplicazioni tra due numeri inseriti dall'utente, contando quante moltiplicazioni vengono effettuate in 10 secondi. Quando il timer di 10 secondi scade, il programma stampa il numero di moltiplicazioni effettuate e termina.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

volatile long int conteggio = 0;

void stop_timer(int signum) {
    printf("Tempo scaduto! Hai effettuato %ld moltiplicazioni.\n", conteggio);
    _exit(0);
}

int main() {
    int a, b, risultato;

    printf("Inserisci due numeri da moltiplicare (es: 5 6): ");
    scanf("%d %d", &a, &b);

    signal(SIGALRM, stop_timer);

    alarm(10);

    while (1) {
        risultato = a * b;
        conteggio++;
    }

    return 0;
}
```

La funzione stop\_timer viene chiamata quando il timer scade (segnale SIGALRM). Stampa il numero di moltiplicazioni effettuate e termina il programma.

La funzione main dopo aver chiesto all'utente di inserire due numeri., imposta il gestore del segnale SIGALRM per chiamare la funzione stop\_timer.

### 9.7.2 Esempio 10.7

Il codice seguente implementa una funzione sleep1 che emula un comportamento simile a sleep, ma utilizzando il segnale SIGALRM per sospendere il processo e poi sveglierlo quando il timer scade.

---

```
#include    <signal.h>
#include    <unistd.h>

static void
sig_alarm(int signo)
{
    /* nothing to do, just return to wake up the pause */
}

unsigned int
sleep1(unsigned int seconds)
{
    if (signal(SIGALRM, sig_alarm) == SIG_ERR)
        return(seconds);
    alarm(seconds);      /* start the timer */
    pause();            /* next caught signal wakes us up */
    return(alarm(0));   /* turn off timer, return unslept time */
}
```

---

La funzione sleep1 accetta un parametro seconds che indica per quanti secondi il processo dovrebbe "dormire". All'inizio, viene impostato il gestore del segnale SIGALRM utilizzando la funzione signal. La funzione alarm(seconds) imposta un timer che invierà un segnale SIGALRM dopo il numero di secondi specificato in seconds. La funzione pause() sospende il processo fino a quando non riceve un segnale che può essere gestito. In questo caso, il processo è sospeso fino a quando non riceve il segnale SIGALRM, che è inviato dal timer impostato da alarm. Il processo disabilita il timer (impostando alarm(0)) e restituisce il tempo che non è stato "dormito", nel caso in cui il processo fosse stato svegliato prima che scadesse il timer. Questo valore viene restituito dalla funzione sleep1.

## 9.8 Signal Sets

Per rappresentare più segnali, ovvero un set di segnali, è necessario introdurre un nuovo tipo di dato: **sigset \_ t**. I signal set sono utilizzati principalmente per manipolare e controllare segnali nel contesto di blocco, sblocco, o gestione dei segnali. Di seguito le cinque funzioni utilizzate per manipolare i set di segnali:

```
#include <signal.h>

int sigemptyset(sigset_t *set);      //Inizializza un signal set vuoto
int sigfillset(sigset_t *set);       //Inizializza un signal set
                                         includendo tutti i segnali
int sigaddset(sigset_t *set, int signo); //Aggiunge un segnale specifico al set
int sigdelset(sigset_t *set, int signo); //Rimuove un segnale specifico dal set
```

Restituiscono tutte 0 in caso di successo e -1 in caso di errore

```
int sigismember(const sigset_t *set, int signo); //Verifica se un segnale
                                                 specifico è presente nel set
```

Restituisce 1 se vero, 0 se falso e -1 in caso di errore

## 9.9 Funzione sigprocmask

Un processo può esaminare la sua signal mask, modificarla o può effettuare entrambe le operazioni in uno step per mezzo di questa funzione. La funzione sigprocmask consente di bloccare o sbloccare segnali modificando la signal mask del processo. La signal mask è un insieme di segnali che un processo blocca attualmente.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oset);
```

Restituisce 0 in caso di successo e -1 in caso di errore

Tale funzione richiede come parametri: un puntatore al signal set che specifica i segnali da bloccare/sbloccare (*set*), un puntatore a un signal set che memorizza la maschera precedente (può essere NULL se non serve) (*oset*) e il parametro *how* che specifica l'azione da eseguire:

- SIG\_BLOCK: Aggiunge i segnali di set alla maschera di segnali bloccati.
- SIG\_UNBLOCK: Rimuove i segnali di set dalla maschera di segnali bloccati

- **SIG\_SETMASK**: Sostituisce la maschera corrente con i segnali in set

Se *set* è un puntatore nullo, la signal mask di un processo non viene modificato e *how* viene ignorato.

### 9.9.1 Esempio 10.14

Nel seguente codice sorgente è stata implementata una funzione che stampa i nomi dei segnali contenuti nella signal mask del processo chiamante:

---

```
#include "apue.h"
#include <errno.h>

void
pr_mask(const char *str)
{
    sigset_t      sigset;
    int          errno_save;

    errno_save = errno;      /* we can be called by signal handlers */
    if (sigprocmask(0, NULL, &sigset) < 0) {
        err_ret("sigprocmask error");
    } else {
        printf("%s", str);
        if (sigismember(&sigset, SIGINT))
            printf(" SIGINT");
        if (sigismember(&sigset, SIGQUIT))
            printf(" SIGQUIT");
        if (sigismember(&sigset, SIGUSR1))
            printf(" SIGUSR1");
        if (sigismember(&sigset, SIGALRM))
            printf(" SIGALRM");

        /* remaining signals can go here */
        printf("\n");
    }
    errno = errno_save;      /* restore errno */
}
```

---

Inizialmente possiamo notare *errno\_save = errno*, necessario perché la funzione può essere chiamata da un gestore di segnali. Se la funzione modifica *errno*, ciò potrebbe interferire con il comportamento di altre parti del programma.

La chiamata a **sigprocmask** con il parametro *how* impostato a 0 (nessuna modifica) consente di ottenere la maschera corrente nella variabile *sigset*.

La funzione **sigismember** controlla se un determinato segnale (ad esempio, SIGINT) appartiene alla maschera dei segnali. Se appartiene, stampa il nome del segnale.

Alla fine viene ripristinato *errno*.

## 9.10 Funzione sigpending

La funzione `sigpending` consente di ottenere l'insieme dei segnali che sono stati inviati al processo ma sono attualmente in stato di pending (in attesa) perché sono bloccati.

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Restituisce 0 in caso di successo e -1 in caso di errore

Il parametro *set* è un puntatore a una variabile di tipo `sigset_t` che, al termine della funzione, conterrà l'insieme dei segnali in stato di pending.

### 9.10.1 Esempio 10.15

Il seguente esempio mostra alcune caratteristiche dei segnali che sono state viste.

Inizialmente il gestore di segnali `sig_quit` viene registrato per intercettare il segnale SIGQUIT. Quando il segnale SIGQUIT viene intercettato, il gestore personalizzato stampa un messaggio e ripristina il comportamento predefinito per SIGQUIT. Il comportamento predefinito di SIGQUIT è la terminazione del processo con generazione di un file di core.

In seguito con la funzione `sigprocmask`, SIGQUIT viene aggiunto alla maschera dei segnali bloccati. La maschera precedente *oldmask* viene salvata per poter essere ripristinata in seguito. Durante i 5 secondi di pausa, qualsiasi tentativo di inviare SIGQUIT al processo lo metterà in stato pending (attesa), poiché il segnale è bloccato.

La funzione `sigpending` viene utilizzata per controllare se SIGQUIT è tra i segnali in attesa. Infine la funzione `sigprocmask` ripristina la maschera dei segnali al suo stato precedente, rimuovendo il blocco su SIGQUIT. Se SIGQUIT era in stato pending, viene gestito immediatamente.

Dopo il ripristino del comportamento predefinito per SIGQUIT, se il segnale viene inviato di nuovo durante `sleep(5)`, il programma termina e crea un file di core.

```
#include "apue.h"

static void sig_quit(int);

int
main(void)
{
    sigset_t    newmask, oldmask, pendmask;

    if (signal(SIGQUIT, sig_quit) == SIG_ERR)
        err_sys("can't catch SIGQUIT");

    /*
     * Block SIGQUIT and save current signal mask.
     */
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    sleep(5); /* SIGQUIT here will remain pending */

    if (sigpending(&pendmask) < 0)
        err_sys("sigpending error");
    if (sigismember(&pendmask, SIGQUIT))
        printf("\nSIGQUIT pending\n");

    /*
     * Restore signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    printf("SIGQUIT unblocked\n");

    sleep(5); /* SIGQUIT here will terminate with core file */
    exit(0);
}

static void
sig_quit(int signo)
{
    printf("caught SIGQUIT\n");
    if (signal(SIGQUIT, SIG_DFL) == SIG_ERR)
        err_sys("can't reset SIGQUIT");
}
```

Esempio di output, dal quale si può notare che i segnali non si accodano:

```
$ ./a.out
^\
SIGQUIT pending                         generate signal once (before 5 seconds are up)
caught SIGQUIT                          after return from sleep
SIGQUIT unblocked                      in signal handler
^\"quit(coredump)                      after return from sigprocmask
                                         generate signal again

$ ./a.out
^\"\"\"\"^\"\"\"^\"\"\"^\"\"\"^\"\"\"^\"\" \
SIGQUIT pending                         generate signal 10 times (before 5 seconds are up)
caught SIGQUIT                          signal is generated only once
SIGQUIT unblocked                      ^\"\"\"quit(coredump)        generate signal again
```

## 9.11 Funzione sigaction

La funzione `sigaction` è un metodo avanzato e più robusto rispetto a `signal` per impostare o modificare un gestore di segnali in programmi C.

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

Restituisce 0 in caso di successo e -1 in caso di errore

La funzione accetta come parametri: *signo* che è il numero del segnale da gestire, *act* che è un puntatore ad un astruttura `struct sigaction` che specifica il nuovo comportamento del segnale e *oact* che è un puntatore a una struttura `struct sigaction` dove verrà salvato il comportamento precedente (se non è NULL).

La struttura `sigaction` definisce il comportamento associato a un segnale:

```
struct sigaction {

    void (*sa_handler)(int);          /* Puntatore al gestore di segnale */

    void (*sa_sigaction)(int, siginfo_t *, void *); /* Gestore esteso */

    sigset_t sa_mask;                /* Maschera di segnali da bloccare */

    int sa_flags;                    /* Flag per comportamento aggiuntivo */

    void (*sa_restorer)(void);       /* Non usato (storico) */

};
```

### 9.11.1 Esempio sigusr1.c

Questo programma utilizza la funzione `sigaction` per gestire il segnale SIGUSR1 e mantiene un conteggio di quante volte il segnale è stato ricevuto durante l'esecuzione del processo:

Inizialmente viene dichiarata un avariabile atomica `sig_atomic_t`, un tipo di dato garantito per essere aggiornato in modo sicuro e atomico durante l'esecuzione di un gestore di segnali (in questo caso viene considerato il segnale SIGUSR1).

Con la funzione `handler` viene definito il gestore del segnale che incrementa semplicemente il

contatore globale `sigusr1_count` ogni volta che SIGUSR1 viene ricevuto.

A questo punto viene configurata **singaction**: la struttura `sigaction` viene inizializzata a zero con `memset`, il campo `sa_handler` viene impostato per puntare al gestore handler e infine la funzione `sigaction` associa il segnale SIGUSR1 al gestore personalizzato.

Infine viene aggiunto un ciclo che continua a funzionare fino a che il tempo trascorso non supera il valore del timeout. Ricordiamo che `time(NULL)` restituisce il timestamp attuale con riferimento all'epoch.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

sig_atomic_t sigusr1_count = 0;

void handler(int signal_number) {
    ++sigusr1_count;
}

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);

    time_t start_time = time(NULL);
    time_t timeout = 250; // Timeout di 250 secondi

    printf("In attesa di segnali per %ld secondi...\n", timeout);

    while (time(NULL) - start_time < timeout) {
        pause(); // Attende segnali
    }

    printf("SIGUSR1 è stato ricevuto %d volte\n", sigusr1_count);
    return 0;
}
```

## 9.12 Funzione `sigsuspend`

La funzione `sigsuspend` è utilizzata in programmi C per sospendere l'esecuzione del processo fino a quando non arriva un segnale che cambia il suo stato. È particolarmente utile quando si

desidera attendere un segnale specifico e modificare temporaneamente la maschera dei segnali durante questa attesa.

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

Non restituisce mai 0, perché viene interrotta solo quando un segnale viene ricevuto. Se termina con un errore, restituisce -1 e imposta errno.

L'argomento *sigmask* è un puntatore a una maschera di segnali (sigset\_t). Durante l'attesa, i segnali specificati in questa maschera saranno sbloccati. Quando `sigsuspend` termina (dopo aver ricevuto un segnale), la maschera originale viene ripristinata automaticamente.

### 9.12.1 Esempio 10.22

Il seguente esempio mostra come è possibile proteggere una regione critica da uno specifico segnale. La funzione **pr\_mask** non è definita ma stampa la maschera dei segnali attualmente attivi utilizzando `sigprocmask` per stampare i segnali attivi. A questo punto `signal(SIGINT, sig_int)` associa il gestore personalizzato `sig_int` al segnale SIGINT (generato da Ctrl+C).

Vengono definite le due maschere di segnali: *waitmask*, la maschera temporanea, che consente solo il segnale SIGUSR1 e *newmask* che blocca SIGINT e verrà applicata alla sezione critica del codice. Con la funzione **sigprocmask** viene bloccato il segnale SIGINT e salva la vecchia maschera dei segnali.

Dopo aver stampato la maschera dei segnali attivi mentre si è nella sezione critica, con la funzione **pr\_mask**, la funzione **sigsuspend** applica temporaneamente la maschera *waitmask* che consente solo il segnale SIGUSR1. Sospende l'esecuzione del programma finché non arriva un segnale. Quando arriva un segnale, il gestore corrispondente viene eseguito. Dopo che il gestore del segnale viene completato, `sigsuspend` termina e ripristina automaticamente la vecchia maschera dei segnali.

Alla fine viene ripristinata la maschera originale dei segnali (*oldmask*), sbloccando il segnale SIGINT.

---

```

#include "apue.h"
static void sig_int(int);
int
main(void)
{
    sigset_t    newmask, oldmask, waitmask;
    pr_mask("program start: ");
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    sigemptyset(&waitmask);
    sigaddset(&waitmask, SIGUSR1);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGINT);

    /*
     * Block SIGINT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    /*
     * Critical region of code.
     */
    pr_mask("in critical region: ");

    /*
     * Pause, allowing all signals except SIGUSR1.
     */
    if (sigsuspend(&waitmask) != -1)
        err_sys("sigsuspend error");
    pr_mask("after return from sigsuspend: ");

    /*
     * Reset signal mask which unblocks SIGINT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");

    /*
     * And continue processing ...
     */
    pr_mask("program exit: ");
    exit(0);
}
static void
sig_int(int signo)
{
    pr_mask("\nin sig_int: ");
}

```

---

Esempio di output:

```

$ ./a.out
program start:
in critical region: SIGINT
^C                                         type the interrupt character
in sig_int: SIGINT SIGUSR1
after return from sigsuspend: SIGINT
program exit:

```

## 9.12.2 Esempio 10.23

Questo programma implementa un ciclo principale che utilizza una combinazione di gestione dei segnali e maschere per attendere in modo sicuro l'arrivo del segnale SIGQUIT.

---

```
#include "apue.h"
volatile sig_atomic_t quitflag; /* set nonzero by signal handler */

static void
sig_int(int signo) /* one signal handler for SIGINT and SIGQUIT */
{
    if (signo == SIGINT)
        printf("\ninterrupt\n");
    else if (signo == SIGQUIT)
        quitflag = 1; /* set flag for main loop */
}

int
main(void)
{
    sigset(SIGINT, sig_int);
    if (signal(SIGINT, sig_int) == SIG_ERR)
        err_sys("signal(SIGINT) error");
    if (signal(SIGQUIT, sig_int) == SIG_ERR)
        err_sys("signal(SIGQUIT) error");

    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGQUIT);

    /* Block SIGQUIT and save current signal mask.
     */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");

    while (quitflag == 0)
        sigsuspend(&zeromask);

    /* SIGQUIT has been caught and is now blocked; do whatever.
     */
    quitflag = 0;

    /* Reset signal mask which unblocks SIGQUIT.
     */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

---

Esempio di output:

```
$ ./a.out
^c                      type the interrupt character
interrupt
^c                      type the interrupt character again
interrupt
^c                      and again
interrupt
^\$                     now terminate with the quit character
```

### 9.12.3 Esempio 10.24

Nei seguenti esempi viene mostrato come i segnali possano essere usati per sincronizzare un PARENT e un CHILD.

```
#include "apue.h"

static volatile sig_atomic_t sigflag; /* set nonzero by sig handler */
static sigset_t newmask, oldmask, zeromask;

static void
sig_usr(int signo) /* one signal handler for SIGUSR1 and SIGUSR2 */
{
    sigflag = 1;
}
```

Inizialmente vengono dichiarate delle variabili globali: la prima funziona come un flag per indicare che un segnale è stato ricevuto, mentre le altre sono strutture utilizzate per gestire le maschere dei segnali.

Il gestore **sig\_usr** è utilizzato per entrambi i segnali e quando il programma riceve uno di questi segnali, il gestore imposta **sigflag** a 1.

```
void
TELL_WAIT(void)
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR1) error");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        err_sys("signal(SIGUSR2) error");
    sigemptyset(&zeromask);
    sigemptyset(&newmask);
    sigaddset(&newmask, SIGUSR1);
    sigaddset(&newmask, SIGUSR2);

    /* Block SIGUSR1 and SIGUSR2, and save current signal mask */
    if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
        err_sys("SIG_BLOCK error");
}
```

Tale funzione: imposta i gestori per i segnali SIGUSR1 e SIGUSR2; inizializza le maschere di segnali e infine blocca SIGUSR1 e SIGUSR2 salvando le vecchie maschere.

La funzione **TELL\_PARENT** usa la funzione **kill** per inviare il segnale SIGUSR2 al processo identificato da pid. Generalmente dopo che il figlio ha terminato la sua operazione, chiama **TELL\_PARENT(getppid())** per notificare al padre.

Nella funzione **WAIT\_PARENT**, la funzione **sigsuspend** fa sì che il processo entri in uno stato di attesa fino a quando non riceve un segnale. Temporaneamente utilizza **zeromask** (una

```

void
TELL_PARENT(pid_t pid)
{
    kill(pid, SIGUSR2);      /* tell parent we're done */
}
void
WAIT_PARENT(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* and wait for parent */
    sigflag = 0;
    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

```

maschera vuota che non blocca alcun segnale) per sospendere l'esecuzione. Una volta ricevuto il segnale, sigsuspend termina e il controllo ritorna al ciclo. Dopo l'attesa, la maschera dei segnali viene ripristinata allo stato precedente.

Allo stesso modo per il CHILD:

```

void
TELL_CHILD(pid_t pid)
{
    kill(pid, SIGUSR1);      /* tell child we're done */
}
void
WAIT_CHILD(void)
{
    while (sigflag == 0)
        sigsuspend(&zeromask); /* and wait for child */
    sigflag = 0;
    /* Reset signal mask to original value */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
}

```

---

## 9.13 Nomi e Numeri del segnale

La funzione strsignal converte il numero di un segnale in una stringa leggibile che rappresenta il nome o la descrizione del segnale.

```

#include <string.h>

char *strsignal(int signo);

```

# Capitolo 10

## Threads

Il concetto di processo è associato, ma comunque distinto da quello di thread con cui si intende l'unità granulare in cui un processo può essere suddiviso (sottoprocesso) e che può essere eseguito a divisione di tempo o in parallelo ad altri thread da parte del processore. In altre parole, un thread è una parte del processo che viene eseguita in maniera concorrente ed indipendente internamente allo stato generale del processo stesso. Il termine inglese rende bene l'idea, in quanto si rifà visivamente al concetto di fune composta da vari fili attorcigliati: se la fune è il processo in esecuzione, allora i singoli fili che la compongono sono i thread. Un processo ha sempre almeno un thread (se stesso), ma in alcuni casi un processo può avere più thread che vengono eseguiti in parallelo. Una differenza sostanziale fra thread e processi consiste nel modo con cui essi condividono le risorse: mentre i processi sono di solito fra loro indipendenti, utilizzando diverse aree di memoria, al contrario i thread di un processo tipicamente condividono le medesime informazioni di stato, la memoria ed altre risorse di sistema; questo significa ad esempio che qualsiasi modifica effettuata sul parent interesserà anche child poiché condividono lo stesso spazio di memoria, a differenza di quella che fa il fork(). Tutti i thread all'interno di un singolo processo hanno accesso agli stessi componenti del processo, come descrittori di file e memoria.

## 10.1 Thread Identification

Proprio come ogni processo ha un ID di processo, ogni thread ha un ID di thread. A differenza dell'ID del processo, che è univoco nel sistema, l'ID del thread ha significato solo nel contesto del processo a cui appartiene. Ricordiamo che un ID di processo, rappresentato dal tipo di dati pid\_t, è un numero intero non negativo. Un ID thread è rappresentato dal tipo di dati pthread\_t. Pertanto, è necessario utilizzare una funzione per confrontare due ID thread.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Restituiscono non zero se sono uguali e zero se sono uguali

Un thread può ottenere il proprio ID thread chiamando la funzione pthread\_self:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Questa funzione può essere utilizzata con pthread\_equal quando un thread deve identificare strutture di dati contrassegnate con il suo ID thread.

## 10.2 Thread Creation

Con pthreads, anche quando un programma viene eseguito, inizia come un singolo processo con un singolo thread di controllo (main threads). Durante l'esecuzione del programma è possibile creare thread aggiuntivi chiamando la funzione pthread\_create.

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tidp,
                  const pthread_attr_t *restrict attr,
                  void *(*start_rtn)(void *, void *restrict arg);
```

La posizione di memoria a cui punta tidp viene impostata sull'ID del thread appena creato quando pthread\_create restituisce correttamente. L'argomento attr viene utilizzato per personalizzare vari attributi del thread. Il thread appena creato inizia l'esecuzione all'indirizzo della

funzione `start_rtn`. Questa funzione accetta un singolo argomento, `arg`, che è un puntatore senza tipo. Se ne hai bisogno passi più di un argomento alla funzione `start_rtn`, quindi devi memorizzarli in un file struttura e passare l'indirizzo della struttura in `arg`. Se `arg` è un intero allora si può effettuare un cast del tipo `(void*)` e poi effettuare nuovamente il cast ad `(int)` all'interno della funzione.

### 10.2.1 Esempio 11.2

Il seguente programma dimostra la creazione di un nuovo thread:

---

```
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t      pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
           (unsigned long)tid, (unsigned long)tid);
}

void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int      err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

---

La funzione `printids()` stampa gli ID di processo e di thread; in particolare tramite la funzione `getpid()` si ottiene il pid del processo, che quindi sarà uguale per entrambe le chiamate, e la funzione `pthread_self()` che permette di ottenere l'ID del thread, che dunque sarà diverso nelle due chiamate. Infine viene stampato il pid e l'ID del thread sia come intero che come esadecimale.

La funzione *thr\_fn()* chiama `printids("new thread: ")`; per stampare le informazioni del thread appena creato e poi ritorna un puntatore nullo `((void *)0)`.

Nella funzione *main()* viene creato il nuovo thread con `pthread_create` che accetta come argomenti: una variabile di tipo `pthread_t` che rappresenta il nuovo thread; `NULL` che specifica gli attributi del thread; il puntatore alla funzione che verrà eseguita nel nuovo thread; `NULL` che indica l'argomento passato alla funzione *thr\_fn* e che in questo caso è nessuno.

## 10.3 Thread Termination

Un singolo thread può uscire in tre modi, interrompendo così il flusso di controllo, senza terminare l'intero processo:

- Il thread può semplicemente ritornare dalla routine di avvio.
- Il thread può essere annullato da un altro thread nello stesso processo.
- Il thread può chiamare `pthread_exit`.

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
```

L'argomento `rval_ptr` è un puntatore senza tipo che rappresenta il valore di ritorno del thread. Questo valore può essere recuperato da un altro thread che chiama `pthread_join`.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **rval_ptr);
```

Restituisce 0 in caso di successo

Questa funzione viene utilizzata per attendere la terminazione di un thread specifico e serve a sincronizzare il thread chiamante con il thread specificato, garantendo che il thread chiamato termini prima che il thread chiamante continui. Se il thread specificato termina con `pthread_exit` o con un `return`, il valore restituito può essere recuperato tramite `rval_ptr`.

L'argomento `pthread_t thread` è l'ID del thread che il chiamante vuole attendere, mentre `void`

*\*\*retval* è un puntatore a una variabile dove verrà memorizzato il valore restituito dal thread (tramite `pthread_exit` o `return`).

Una thread può richiedere che un'altra thread nello stesso processo venga cancellata:

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

Restituisce 0 in caso di successo e un altro numero in caso di errore.

### 10.3.1 Esempio 11.3

Il seguente programma mostra come recuperare il codice di uscita da un thread che è terminato.

---

```
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}

int
main(void)
{
    int          err;
    pthread_t    tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

---

Come possiamo vedere, quando un thread esce chiamando `pthread_exit` o semplicemente ritornando dalla routine di avvio, lo stato di uscita può essere ottenuto da un altro thread chiamando `pthread_join`. Questa viene chiamata per ciascun thread (tid1 e tid2). Blocca il thread chiamante (probabilmente il thread principale) finché il thread specificato non termina. Dopo che `pthread_join` ha atteso la terminazione di un thread, il valore di ritorno di quel thread viene memorizzato in `tret`.

### 10.3.2 Esempio `thread_incd.c`

Nel seguente codice sorgente è stato implementato un programma multithread.

Nella funzione `threadFunc` nel seguente codice è stato implementato un ciclo for nel quale viene incrementato *arg* volte il valore di *glob*. *Glob* è una variabile condivisa tra più thread e ogni thread legge il valore corrente e lo incrementa di 1.

```
#include <pthread.h>
#include "tlpi_hdr.h"

static volatile int glob = 0; /* "volatile" prevents compiler optimizations
                             of arithmetic operations on 'glob' */
/* Loop 'arg' times incrementing 'glob' */
static void *
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j;

    for (j = 0; j < loops; j++) {
        loc = glob;
        loc++;
        glob = loc;
    }

    return NULL;
}
```

Nel main del codice sorgente viene specificato che Il numero di loop (ovvero quante volte viene incrementato *glob*) può essere passato come argomento da riga di comando. Se non specificato, assume un valore predefinito di 10,000,000.

A questo punto due thread (t1 e t2) vengono creati con `pthread_create()` e dopo la creazione, `pthread_join()` viene utilizzato per garantire che il main thread attenda la terminazione di entrambi i thread.

```

int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}

```

## 10.4 Thread Synchronization

Quando più thread di controllo condividono la stessa memoria, dobbiamo assicurarcene che ogni thread vede una visualizzazione coerente dei propri dati. Se ogni thread utilizza variabili, dall'altro thread non verranno letti o modificati e quindi non esisteranno problemi di coerenza. Allo stesso modo, se una variabile è di sola lettura, non vi è alcun problema di coerenza con più di un thread che leggono il suo valore allo stesso tempo. Tuttavia, quando un thread può modificare una variabile e gli altri thread possono leggere o modificare, dobbiamo sincronizzare i thread. Quando un thread modifica una variabile, altri thread possono potenzialmente vedere incoerenze durante la lettura del valore di quella variabile.

### 10.4.1 Mutexes

Un mutex (abbreviazione di mutual exclusion, esclusione mutua) è un meccanismo utilizzato nei programmi multithread per sincronizzare l'accesso a risorse condivise, come variabili o strutture dati. Serve a garantire che solo un thread alla volta possa accedere alla risorsa protetta dal mutex. In altre parole, un mutex è come un lucchetto:

- Quando un thread vuole accedere a una risorsa condivisa, deve "prendere il lucchetto"

(bloccare il mutex)

- Una volta finito, il thread deve "rilasciare il lucchetto" (sbloccare il mutex), permettendo ad altri thread di accedere alla risorsa.

Una variabile mutex è rappresentata da **pthread\_mutex\_t** data type. Prima di usarlo, il mutex deve essere inizializzato e lo si può fare settando il mutex alla costante PTHREAD\_MUTEX\_INITIALIZER, oppure chiamando la seguente funzione:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Per inizializzare un mutex con gli attributi di default possiamo settare *attr* a NULL.

Un thread blocca il mutex prima di accedere alla risorsa condivisa usando `pthread_mutex_lock()`.

Se un altro thread ha già bloccato il mutex, il thread corrente aspetterà finché il mutex non viene sbloccato.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Dopo aver terminato l'uso della risorsa condivisa, il thread sblocca il mutex con `pthread_mutex_unlock()`, permettendo ad altri thread di accedere alla risorsa

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Quando il mutex non è più necessario, viene distrutto con `pthread_mutex_destroy()`.

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

## Esempio `thread_incr_mutex.c`

Il seguente codice è una versione aggiornata dell'esempio precedente, che utilizza un mutex per sincronizzare l'accesso alla variabile globale `glob`, garantendo la correttezza nei programmi multithread.

```
#include <pthread.h>
#include "tlpi_hdr.h"

static volatile int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                  /* Loop 'arg' times incrementing 'glob' */
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}
```

Come spiegato precedentemente `glob` è una variabile globale condivisa tra i thread. Il modificatore `volatile` informa il compilatore che `glob` può essere modificata da qualcosa di esterno al flusso di esecuzione del programma, come un altro thread. Tuttavia, `volatile` da solo non garantisce la sincronizzazione, da cui la necessità di un mutex. Con la costante `PTHREAD_MUTEX_INITIALIZER` viene inizializzato un mutex.

Nell'implementazione del loop che viene fatto eseguire da ogni thread, prima di accedere a `glob`, il thread blocca il mutex usando `pthread_mutex_lock` e se un altro thread ha già bloccato il mutex, il thread corrente aspetta finché non viene sbloccato. Dopo aver completato l'accesso a `glob`, il thread sblocca il mutex, permettendo ad altri thread di accedere con la funzione `pthread_mutex_unlock`.

## 10.4.2 Reader-Writer Locks

Un reader-writer lock (blocco lettore-scrittore) è un meccanismo di sincronizzazione utilizzato nei programmi multithread per ottimizzare l'accesso a risorse condivise, specialmente quando ci sono molte letture e poche scritture.

Con un reader-writer lock, si possono soddisfare due condizioni:

- Lettori multipli possono accedere alla risorsa contemporaneamente, fintanto che non ci sono scrittori
- Uno scrittore ha accesso esclusivo alla risorsa, bloccando sia altri scrittori che i lettori

Una variabile reader-writer è rappresentata da **pthread\_rwlock\_t** data type. Prima di usarlo, il reader-writer deve essere inizializzato e lo si può fare settandolo alla costante PTH-READ\_RWLOCK\_INITIALIZER, oppure chiamando la seguente funzione:

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                      const pthread_rwlockattr_t *restrict attr);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Per inizializzare un reader-writer con gli attributi di default possiamo settare *attr* a NULL.

Un thread può bloccare il lock per lettura usando `pthread_rwlock_rdlock()`

```
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Un thread può bloccare il lock per scrittura usando `pthread_rwlock_wrlock()`

```
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Usare `pthread_rwlock_unlock()` per rilasciare il lock:

```
#include <pthread.h>
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

Quando il lock non è più necessario, distruggerlo con `pthread_rwlock_destroy()`:

```
#include <pthread.h>
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

### 10.4.3 Reader-Writer Locking with Timeouts

Il reader-writer locking con timeout consente ai thread di attendere un periodo limitato per ottenere il lock (sia in lettura che in scrittura). Se il lock non è disponibile entro il tempo specificato, la funzione restituisce un errore invece di bloccare il thread indefinitamente.

In POSIX, questa funzionalità è supportata usando le funzioni:

```
#include <pthread.h>
#include <time.h>
int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tspt);
int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                               const struct timespec *restrict tspt);
```

Restituiscono 0 in caso di successo ed un numero errore in caso di fallimento

### 10.4.4 Condition Variables

Le condition variables (variabili di condizione) sono meccanismi di sincronizzazione usati nei programmi multithread per consentire ai thread di comunicare e coordinarsi in modo efficiente.

Insieme a un mutex, permettono a un thread di attendere fino a quando una certa condizione non diventa vera, evitando cicli di polling continuo che consumano risorse.

Flusso tipico di utilizzo di una condition variable:

- Il thread blocca il mutex: Prima di controllare o modificare la variabile condivisa, il thread blocca il mutex per assicurare un accesso esclusivo
- Controllo dello stato della variabile condivisa: Il thread verifica lo stato della variabile condivisa per determinare se può proseguire
- Attesa se lo stato non è desiderato: Se la variabile condivisa non è nello stato desiderato il thread rilascia il mutex per consentire ad altri thread di modificare la variabile condivisa. Successivamente chiama `pthread_cond_wait()` per andare in attesa sulla condition variable finché non viene notificato da un altro thread
- Risveglio e blocco del mutex: Quando il thread viene risvegliato (a seguito di un segnale o broadcast), riacquisisce automaticamente il mutex.

La funzione `pthread_cond_init` inizializza una condition variable:

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
```

Restituiscono 0 in caso di successo ed un numero errore in caso di fallimento

La funzione `pthread_cond_destroy` distrugge una condition variable:

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

Restituiscono 0 in caso di successo ed un numero errore in caso di fallimento

La funzione `pthread_cond_wait` mette un thread in attesa fino a quando la condition variable viene segnalata. Quando viene chiamata questa funzione:

- Il thread rilascia il mutex associato alla variabile condivisa. In questo modo, altri thread possono modificare la variabile condivisa mentre il thread corrente è in attesa
- Il thread va in attesa (in stato di sleep) sulla condition variable
- Quando il thread viene risvegliato (grazie a un segnale o broadcast sulla condition variable), riacquisisce automaticamente il mutex per garantire un accesso sicuro alla variabile condivisa

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

Restituiscono 0 in caso di successo ed un numero errore in caso di fallimento

La funzione pthread\_cond\_broadcast sveglia tutti i thread in attesa sulla condition variable:

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Restituisce 0 in caso di successo ed un numero errore in caso di fallimento

La funzione pthread\_cond\_signal sveglia un singolo thread in attesa sulla condition variable

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

### Esempio prod\_no\_condvar.c

Il codice seguente implementa un thread produttore che produce unità a intervalli regolari (1 secondo ciascuno) e aggiorna la variabile condivisa avail per segnalare che un'unità è disponibile. La funzione **producer** produce un certo numero di unità (cnt), specificato dall'argomento arg. Il valore di avail viene incrementato di 1 per ogni unità prodotta. Questa operazione è protetta da un mutex e dopo aver modificato avail, il produttore sblocca il mutex con pthread\_mutex\_unlock.

```

#include <time.h>
#include <pthread.h>
#include <stdbool.h>
#include "tlpi_hdr.h"

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static int avail = 0;

static void *
producer(void *arg)
{
    int cnt = atoi((char *) arg);

    for (int j = 0; j < cnt; j++) {
        sleep(1);

        /* Code to produce a unit omitted */

        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        avail++;          /* Let consumer know another unit is available */

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}

```

Nel main all'inizio viene memorizzata l'ora corrente in t per tenere traccia del tempo trascorso durante la produzione e il consumo delle unità.

A questo punto un ciclo viene utilizzato per creare ogni thread produttore specificato da argv[j].

Ogni produttore incrementa la variabile condivisa avail ogni volta che produce un'unità.

Un ciclo infinito consuma tutte le unità disponibili, protetto dal mutex per evitare condizioni di gara. In questo ciclo i thread consumatori eseguono pthread\_mutex\_lock per accedere a avail.

Vengono consumate tutte le unità disponibili. Dopo ogni consumo, il mutex viene sbloccato con pthread\_mutex\_unlock. Se tutte le unità richieste (totRequired) sono state consumate, il ciclo si interrompe (done = true).

```

int
main(int argc, char *argv[])
{
    time_t t = time(NULL);

    int totRequired = 0;           /* Total number of units that all
                                    threads will produce */

    /* Create all threads */

    for (int j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);

        pthread_t tid;
        int s = pthread_create(&tid, NULL, producer, argv[j]);
        if (s != 0)
            errExitEN(s, "pthread_create");

    }

    /* Use a polling loop to check for available units */

    int numConsumed = 0;           /* Total units so far consumed */
    bool done = false;

    for (;;) {
        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        while (avail > 0)           /* Consume all available units */

            /* Do something with produced unit */

            numConsumed++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t),
                   numConsumed);

            done = numConsumed >= totRequired;
        }
        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");

        if (done)
            break;

        /* Perhaps do other work here that does not require mutex lock */
    }

    exit(EXIT_SUCCESS);
}

```

## Esempio prod\_condvar.c

Di seguito il codice precedente implementa un modello produttore-consutatore basato su condition variables.

A differenza del codice precedente, in questo caso viene inizializzata una condition variable utilizzata per notificare ai consumatori che ci sono unità disponibili. Nella funzione **producer** dopo aver prodotto un'unità, incrementa avail e segnala il consumatore usando `pthread_cond_signal()`.

```

#include <time.h>
#include <pthread.h>
#include <stdbool.h>
#include "tlpi_hdr.h"

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static int avail = 0;

static void *
producer(void *arg)
{
    int cnt = atoi((char *) arg);

    for (int j = 0; j < cnt; j++) {
        sleep(1);

        /* Code to produce a unit omitted */

        int s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        avail++; /* Let consumer know another unit is available */

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");

        s = pthread_cond_signal(&cond); /* Wake sleeping consumer */
        if (s != 0)
            errExitEN(s, "pthread_cond_signal");
    }

    return NULL;
}

```

Nel **main** ogni produttore aggiorna la variabile condivisa `avail` e notifica i consumatori tramite `pthread_cond_signal`. Un ciclo infinito (`for(;;)`) viene utilizzato per consumare tutte le unità prodotte. All'interno del ciclo il mutex viene bloccato per accedere a `avail`, se `avail == 0`, il consumatore si mette in attesa con `pthread_cond_wait`. Quando viene notificato tramite `pthread_cond_signal`, consuma le unità disponibili.

```

int
main(int argc, char *argv[])
{
    time_t t = time(NULL);

    int totRequired = 0; /* Total number of units that all threads
                           will produce */
    /* Create all threads */

    for (int j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);

        pthread_t tid;
        int s = pthread_create(&tid, NULL, producer, argv[j]);
        if (s != 0)
            errExitEN(s, "pthread_create");
    }
}

```

```

int numConsumed = 0;           /* Total units so far consumed */
bool done = false;

for (;;) {
    int s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    while (avail == 0) {           /* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
        if (s != 0)
            errExitEN(s, "pthread_cond_wait");
    }

    /* At this point, 'mtx' is locked... */

    while (avail > 0) {           /* Consume all available units */
        /* Do something with produced unit */

        numConsumed++;
        avail--;
        printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t),
               numConsumed);
    }

    done = numConsumed >= totRequired;
}

s = pthread_mutex_unlock(&mtx);
if (s != 0)
    errExitEN(s, "pthread_mutex_unlock");

if (done)
    break;

/* Perhaps do other work here that does not require mutex lock */
}

exit(EXIT_SUCCESS);
}

```

## 10.5 Thread Attributes

Gli attributi di tipo **pthread\_attr\_t** consentono di configurare vari aspetti del comportamento di un thread prima della sua creazione. L'oggetto **pthread\_attr\_t** è utilizzato come parametro nelle funzioni come **pthread\_create**, per specificare caratteristiche come la dimensione dello stack, lo stato di detach, la politica di pianificazione, e altro.

La funzione **pthread\_attr\_init** è utilizzata per inizializzare un oggetto di tipo **pthread\_attr\_t**, che viene utilizzato per specificare gli attributi di un thread in POSIX Threads (pthread). Una volta inizializzato, l'oggetto può essere configurato con attributi personalizzati prima di essere passato a **pthread\_create** per la creazione di un thread.

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

Il parametro *attr* è Un puntatore a un oggetto di tipo `pthread_attr_t` e dopo l'inizializzazione, l'oggetto conterrà i valori predefiniti degli attributi.

La funzione **`pthread_attr_destroy`** libera le risorse allocate per un oggetto attributo:

```
#include <pthread.h>
int pthread_attr_destroy(pthread_attr_t *attr);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

La funzione **`pthread_attr_setdetachstate`** setta lo stato di detach, ovvero determina se il thread sarà joinable (si può attendere con `pthread_join`) o detached (non si può attendere e le risorse vengono liberate automaticamente al termine del thread):

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

Dove *detachstate* può essere:

- `PTHREAD_CREATE_JOINABLE` (default): Thread joinable.
- `PTHREAD_CREATE_DETACHED`: Thread detached.

La funzione **`pthread_attr_getdetachstate`** permette di leggere lo stato attuale:

```
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *restrict attr,
                                int *detachstate);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

La funzione **`pthread_attr_setstack`** permette di specificare manualmente il puntatore e la dimensione dello stack del thread. Lo stack, o pila, è una struttura di memoria utilizzata nei computer per gestire l'esecuzione dei programmi e allocare spazio per le variabili locali, i parametri delle funzioni e i valori di ritorno delle chiamate di funzione.

```

#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                         void **restrict stackaddr,size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                         size_t stacksize);

```

Restituiscono 0 in caso di successo e un numero errore in caso di fallimento

La funzione **pthread\_attr\_setstacksize** specifica la dimensione della memoria dello stack per il thread:

```

#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                             size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);

```

Restituiscono 0 in caso di successo e un numero errore in caso di fallimento

## 10.6 Reentrancy

La reentrancy (o "ri-entranza") è una proprietà di funzioni o blocchi di codice che possono essere eseguiti contemporaneamente da più thread o chiamate, senza conflitti o comportamenti indesiderati. Le funzioni reentrant sono particolarmente importanti nei sistemi multi-threading o in ambienti che richiedono chiamate asincrone.

Una funzione è considerata reentrant se:

- Non utilizza variabili globali o statiche condivise: Ogni thread o chiamata ha una copia indipendente delle variabili utilizzate dalla funzione
- Non modifica strutture di dati condivise: Se accede a risorse condivise, lo fa utilizzando meccanismi di sincronizzazione come mutex o semafori.
- Non si affida a stati persistenti: Lo stato della funzione dipende solo dagli argomenti passati e dalle variabili locali.

- Non utilizza funzioni non thread-safe: Evita chiamate a funzioni che accedono a risorse globali (es. strtok o asctime).

## 10.7 Thread-Specific Data

Il Thread-Specific Data (TSD) è una funzionalità che consente a ciascun thread in un'applicazione multi-threaded di avere una propria copia di dati, indipendentemente dagli altri thread. Questo è utile quando più thread devono accedere a una variabile o struttura dati, ma ogni thread deve operare su una versione separata per evitare conflitti. I passaggi principali sono:

- Creare una chiave: una chiave (pthread\_key\_t) viene utilizzata per identificare i dati specifici di un thread.
- Associare dati alla chiave: ogni thread può associare un valore specifico a questa chiave.
- Accedere ai dati: I thread utilizzano la chiave per accedere ai propri dati specifici.
- Distruggere la chiave: Quando non è più necessaria, la chiave può essere eliminata.

La funzione **pthread\_key\_create** crea una chiave per il TSD:

```
#include <pthread.h>
int pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *));
```

Restituiscono 0 in caso di successo e un numero errore in caso di fallimento

Dove *key* è il puntatore alla chiave da creare e *destructor* è una funzione opzionale chiamata per liberare la memoria associata al TSD quando il thread termina.

La funzione **pthread\_setspecific** associa un valore specifico alla chiave per il thread corrente mentre con **pthread\_getspecific** è possibile recuperare il valore specifico:

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

Restituisce un valore specifico del thread oppure NULL se nessun valore è stato associato con

la key

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

La funzione **pthread\_key\_delete** distrugge una chiave TSD:

```
#include <pthread.h>
int pthread_key_delete(pthread_key_t key);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

### 10.7.1 Esempio ATEST22TLS.c

Il codice è un esempio di utilizzo del Thread-Local Storage (TLS) con la parola chiave `__thread` per garantire che ogni thread abbia una copia indipendente delle variabili `TLS_data1` e `TLS_data2`. `__thread` è un qualificatore specifico di GCC che permette di dichiarare variabili con thread-local storage. Ogni thread ha una copia distinta di `TLS_data1` e `TLS_data2`. Quando un thread modifica queste variabili, le modifiche non influenzano altri thread.

```
#define _MULTI_THREADS
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void foo(void); /* Functions that use the TLS data */
void bar(void);

#define checkResults(string, val) { \
    if (val) { \
        printf("Failed with %d at %s", val, string); \
        exit(1); \
    } \
}

__thread int TLS_data1;
__thread int TLS_data2;

#define NUMTHREADS 8

typedef struct {
    int data1;
    int data2;
} threadparm_t;
```

Nella funzione `theThread` ogni thread assegna i dati di `gData` alle variabili TLS locali. Le variabili TLS vengono poi utilizzate in `foo()` e `bar()`.

```

void *theThread(void *parm)
{
    int             rc;
    threadparm_t   *gData;

    printf("Thread %.16lx: Entered\n", pthread_self());
    gData = (threadparm_t *)parm;

    TLS_data1 = gData->data1;
    TLS_data2 = gData->data2;

    foo();
    return NULL;
}

```

Le funzioni **foo()** e **bar()** stampano i valori di `TLS_data1` e `TLS_data2` per il thread corrente.

```

void foo() {
    printf("Thread %.16lx: foo(), TLS data=%d %d\n",
           pthread_self(), TLS_data1, TLS_data2);
    bar();
}

void bar() {
    printf("Thread %.16lx: bar(), TLS data=%d %d\n",
           pthread_self(), TLS_data1, TLS_data2);
    return;
}

```

Nel **main** ogni thread riceve un puntatore a un elemento dell'array `gData` come parametro. L'elemento contiene i dati specifici per quel thread.

```

int main(int argc, char **argv)
{
    pthread_t        thread[NUMTHREADS];
    int              rc=0;
    int              i;
    threadparm_t     gData[NUMTHREADS];

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Create/start threads\n");
    for (i=0; i < NUMTHREADS; i++) {
        /* Create per-thread TLS data and pass it to the thread */
        gData[i].data1 = i;
        gData[i].data2 = (i+1)*2;
        rc = pthread_create(&thread[i], NULL, theThread, &gData[i]);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait for the threads to complete, and release their resources\n");
    for (i=0; i < NUMTHREADS; i++) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    printf("Main completed\n");
    return 0;
}

```

Esempio di output:

Enter Testcase - ./a.out

```

Create/start threads

Thread 7f9c3c34a700: Entered

Thread 7f9c3c34a700: foo(), TLS data=0 2

Thread 7f9c3c34a700: bar(), TLS data=0 2

Thread 7f9c3c349700: Entered

Thread 7f9c3c349700: foo(), TLS data=1 4

Thread 7f9c3c349700: bar(), TLS data=1 4

...

Wait for the threads to complete, and release their resources

Main completed

```

## 10.8 Cancel Options

La funzione `pthread_setcancelstate` è una delle API offerte da pthread per gestire la cancellazione di un thread. Consente di abilitare o disabilitare la possibilità di cancellare un thread durante la sua esecuzione.

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

Restituisce 0 in caso di successo e ed un numero errore in caso di fallimento

Dove *state* specifica il nuovo stato di cancellazione del thread e può assumere uno dei seguenti valori:

- `PTHREAD_CANCEL_ENABLE`: Abilita la possibilità di cancellare il thread.
- `PTHREAD_CANCEL_DISABLE`: Disabilita la possibilità di cancellare il thread.

Mentre *oldstate*(Facoltativo) è un puntatore a un intero in cui viene memorizzato il precedente stato di cancellazione. Se è NULL, il precedente stato non viene memorizzato.

## 10.9 Thread e Segnali

La funzione **pthread\_sigmask** è una delle funzioni della libreria pthread utilizzate per manipolare le maschere di segnali nei thread. Consente di bloccare o sbloccare segnali in un thread specifico. In generale, i segnali sono usati per notificare eventi asincroni al processo (o ai thread). Con pthread\_sigmask, è possibile controllare quali segnali vengono gestiti o ignorati da un thread.

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

Dove *how* specifica come modificare la maschera dei segnali per il thread e può assumere uno dei seguenti valori: SIG\_BLOCK, SIG\_UNBLOCK e SIG\_SETMASK, *set* è un puntatore a un oggetto sigset\_t che contiene il set di segnali da bloccare, sbloccare o impostare e *oset* è un puntatore a un oggetto sigset\_t in cui verrà memorizzata la maschera precedente dei segnali. La funzione **sigwait** è utilizzata in ambienti multithreaded per attendere sincronamente l'arrivo di uno o più segnali. La funzione consente a un thread di attendere un segnale specifico senza che il segnale venga elaborato immediatamente da un altro thread. Questo è utile per la gestione dei segnali in modo coordinato e per evitare che il programma venga interrotto senza controllo.

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict signop);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

Dove *set* è un puntatore a un oggetto sigset\_t che rappresenta il set di segnali da attendere. Il thread che chiama sigwait verrà bloccato finché uno dei segnali nel set non verrà ricevuto. Mentre *sig* è un puntatore a una variabile intera in cui verrà memorizzato il numero del segnale che ha causato il ritorno dalla funzione. Dopo che sigwait è ritornata, questa variabile contiene il segnale che è stato catturato dal thread.

La funzione **pthread\_kill** è utilizzata per inviare un segnale a un thread specifico all'interno di un processo:

```
#include <signal.h>
int pthread_kill(pthread_t thread, int signo);
```

Restituisce 0 in caso di successo e un numero errore in caso di fallimento

### 10.9.1 Esempio 12.16

Il seguente programma è un esempio di gestione dei segnali in un'applicazione multithreaded, con l'uso di pthread e delle funzioni di gestione dei segnali in un contesto multithreaded.

Inizialmente vengono definite le variabili globali: *quitflag* che il thread *thr\_fn* usa per segnalare quando il segnale SIGQUIT è stato ricevuto e *mask* che è la maschera dei segnali che il thread principale e il thread *thr\_fn* devono bloccare. Include SIGINT e SIGQUIT.

Il thread **thr\_fn** entra in un ciclo infinito, dove attende i segnali tramite sigwait. Se il segnale è SIGINT, stampa un messaggio. Se il segnale è SIGQUIT, imposta quitflag a 1, sblocca il mutex, e poi segnala al thread principale che può proseguire (con pthread\_cond\_signal). In caso di segnali non gestiti, il programma termina con un errore.

La funzione **main** inizia creando una maschera di segnali che include SIGINT e SIGQUIT. Usa pthread\_sigmask per bloccare questi segnali nel thread principale, assicurando che non vengano gestiti subito. Crea un thread (con pthread\_create) che gestirà i segnali attraverso sigwait. Il thread principale quindi attende che quitflag venga impostato a 1, il che significa che SIGQUIT è stato ricevuto e gestito. Dopo che il thread secondario ha gestito il segnale, la funzione main ripristina la maschera originale dei segnali usando sigprocmask, sbloccando così SIGQUIT.

---

```
#include "apue.h"
#include <pthread.h>

int      quitflag; /* set nonzero by thread */
sigset_t  mask;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t waitloc = PTHREAD_COND_INITIALIZER;

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0)
            err_exit(err, "sigwait failed");
        switch (signo) {
        case SIGINT:
            printf("\ninterrupt\n");
            break;
        case SIGQUIT:
            pthread_mutex_lock(&lock);
            quitflag = 1;
            pthread_mutex_unlock(&lock);
            pthread_cond_signal(&waitloc);
            return(0);

        default:
            printf("unexpected signal %d\n", signo);
            exit(1);
        }
    }
}

int
main(void)
{
    int      err;
    sigset_t oldmask;
    pthread_t tid;

    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, &oldmask)) != 0)
        err_exit(err, "SIG_BLOCK error");

    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&waitloc, &lock);
    pthread_mutex_unlock(&lock);

    /* SIGQUIT has been caught and is now blocked; do whatever */
    quitflag = 0;

    /* reset signal mask which unblocks SIGQUIT */
    if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
        err_sys("SIG_SETMASK error");
    exit(0);
}
```

---

## 10.10 Thread e I/O

Le funzioni pread e pwrite sono funzioni di lettura e scrittura di basso livello in C che operano su file descriptor. Sono versioni thread-safe di read e write e sono progettate per essere utilizzate in ambienti multithreaded. Queste sono delle funzioni atomiche, quindi permettono a ogni thread di operare su una specifica porzione del file senza bloccare e senza essere bloccati da altri thread.

# Capitolo 11

## Daemon Processes

I processi demoni (dal termine inglese daemon) sono programmi o processi che girano in background su un sistema operativo. Sono progettati per svolgere compiti specifici senza interazione diretta con l'utente, fornendo servizi o supporto ad altri processi. Solitamente si avviano durante il processo di boot del sistema operativo e rimangono in esecuzione fino allo spegnimento.

### 11.1 Caratteristiche dei Daemon

Il comando **ps** è uno strumento potente per visualizzare informazioni sui processi in esecuzione, inclusi i demoni. Per identificare i demoni in esecuzione, devi cercare processi che operano in background e che spesso non hanno un terminale associato.

Il comando ps -axj è un'opzione avanzata di ps che fornisce una panoramica dettagliata dei processi attivi:

- **-a**: Mostra tutti i processi associati a tutti gli utenti
- **-x**: Include anche i processi che non hanno un terminale associato (come i demoni)
- **-j**: Mostra l'informazione in formato "job format", che evidenzia i dettagli relativi al controllo dei job.

Tutti i processi con un parent process ID uguale a 0 è generalmente un kernel process iniziato come parte della procedura bootstrap del sistema. In un semplice output di ps, i kernel daemons appaiono con i loro nomi tra le parentesi quadre. In particolare esiste uno speciale kernel process *kthreadd* per creare gli altri kernel processe, quindi kthreadd diventa il padre di tutti gli altri processi daemon.

## 11.2 Coding Rules

Alcune regole di base per codificare un demone impediscono il verificarsi di interazioni indesiderate. Dichiariamo qui queste regole e poi mostriamo una funzione, `daemonize`, che le implementa

- Chiama `umask` per impostare la maschera di creazione della modalità file su un valore noto, solitamente 0. La maschera di creazione della modalità file ereditata potrebbe essere impostata per negare determinati permessi. Se il processo demone crea file, potrebbe voler impostare autorizzazioni specifiche. Ad esempio, se crea file con lettura di gruppo e scrittura di gruppo abilitate, una maschera di creazione della modalità file che disattiva una di queste autorizzazioni annullerebbe i suoi sforzi.
- Chiama `fork` e fai uscire il genitore. Questo fa diverse cose. Innanzitutto, se il demone è stato avviato come un semplice comando di shell, la terminazione del genitore fa sì che la shell pensi che il comando sia terminato. In secondo luogo, il figlio eredita l'ID del gruppo di processi del genitore ma ottiene un nuovo ID di processo, quindi è garantito che il bambino non è un leader del gruppo di processo. Questo è un prerequisito per la chiamata a `setsid` che verrà eseguito successivamente.
- Chiama `setsid` per creare una nuova sessione. In questo modo il processo (a) diventa il leader di una nuova sessione, (b) diventa il leader di un nuovo gruppo di processi e (c) viene dissociato dal suo terminale di controllo.
- Cambia la directory di lavoro corrente nella directory root

- I descrittori di file non necessari dovrebbero essere chiusi. Ciò impedisce al demone di tenere aperti eventuali descrittori che potrebbe aver ereditato dal genitore.
- Alcuni demoni aprono i descrittori di file 0, 1 e 2 in /dev/null in modo che qualsiasi routine di libreria che tenta di leggere dallo standard input o di scrivere sullo standard output o l'errore standard non avrà alcun effetto. Questo viene fatto per evitare comportamenti indesiderati: se una routine della libreria tenta di leggere dallo standard input (file descriptor 0), ma non c'è input disponibile, potrebbe bloccare il demone e allo stesso modo scrivere su standard output (file descriptor 1) o standard error (file descriptor 2) senza reindirizzarli può causare messaggi errati in output

### 11.2.1 Esempio 13.1

---

```

#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>

void
daemonize(const char *cmd)
{
    int             i, fd0, fd1, fd2;
    pid_t           pid;
    struct rlimit   rl;
    struct sigaction sa;

    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();

    /*
     * Ensure future opens won't allocate controlling TTYs.
     */
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
}

```

```

sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: can't ignore SIGHUP", cmd);
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);

/*
 * Change the current working directory to the root so
 * we won't prevent file systems from being unmounted.
 */
if (chdir("/") < 0)
    err_quit("%s: can't change directory to /", cmd);

/*
 * Close all open file descriptors.
 */
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);

/*
 * Attach file descriptors 0, 1, and 2 to /dev/null.
 */
fd0 = open("/dev/null", O_RDWR);
fd1 = dup(0);
fd2 = dup(0);

/*
 * Initialize the log file.
 */
openlog(cmd, LOG_CONS, LOG_DAEMON);
if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
    syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
           fd0, fd1, fd2);
    exit(1);
}
}

```

---

## 11.3 Error Logging

Un problema che ha un demone è come gestire i messaggi di errore. Non può semplicemente scrivere un errore standard, poiché non dovrebbe avere un terminale di controllo. Non vogliamo che tutti i demoni scrivano sul dispositivo console e non vogliamo che ogni demone scriva il proprio messaggio di errore in un file separato. E' necessario un daemon centrale per l'error-logging. Un **error logging daemon** è un demone dedicato alla registrazione e gestione dei messaggi di errore, spesso utilizzato per raccogliere, organizzare e archiviare informazioni sui problemi del sistema o delle applicazioni.

La funzione **openlog** è una funzione utilizzata per configurare la registrazione dei messaggi di log:

```
#include <syslog.h>
```

```
void openlog(const char *ident, int option, int facility);
```

Dove il parametro *ident* è una stringa che identifica la sorgente dei messaggi di log. Questo valore verrà aggiunto come prefisso in ogni messaggio di log, quindi tipicamente si usa il nome del programma. Il parametro *facility* specifica la categoria o il contesto del messaggio di log e può essere: LOG\_AUTH (Messaggi relativi all'autenticazione), LOG\_CRON (Messaggi relativi ai processi cron), LOG\_DAEMON (Messaggi relativi ai demoni di sistema), etc. Infine il parametro *option* è un insieme di flag combinati con un operatore OR (|) per specificare opzioni di logging. I valori comuni includono:

- LOG\_CONS: Invia il messaggio di log anche alla console se non è possibile scriverlo nel file di log
- LOG\_NDELAY: Apre immediatamente la connessione al demone di log (syslogd) invece di aspettare il primo messaggio
- LOG\_PID: Include l'ID del processo (PID) nei messaggi di log

La funzione **syslog** è utilizzata per inviare messaggi al sistema di logging centralizzato, come syslogd:

```
#include <syslog.h>
void syslog(int priority, const char *format, ...);
```

La funzione **closelog** chiude la connessione aperta al demone di log da una precedente chiamata a openlog:

```
#include <syslog.h>
void closelog(void);
```

La funzione **setlogmask** consente di filtrare i messaggi di log in base ai loro livelli di priorità, permettendo di specificare quali tipi di messaggi devono essere registrati e quali devono essere ignorati.

```
#include <syslog.h>
int setlogmask(int mask);
```

dove il parametro *mask* è un valore intero che specifica una maschera di bit per abilitare o disabilitare determinati livelli di log.

Le funzioni del sistema di logging (*openlog* e *syslog*) vengono utilizzate per configurare e registrare un messaggio di errore:

```
openlog("lpd", LOG_PID, LOG_LPR);  
syslog(LOG_ERR, "open error for %s: %m", filename);
```

Nella funzione *openlog*: "lpd" è l'identificatore del programma; LOG\_PID è un'opzione che specifica che il PID del processo verrà incluso nei messaggi di log e LOG\_LPR è il parametro facility e indica che i messaggi appartengono alla categoria di logging LPR (Line Printer).

Nella funzione *syslog* LOG\_ERR indica il livello di severità del messaggio e in questo caso, si tratta di un errore, mentre nella stringa specificata che viene stampata, %m è una specifica speciale per *syslog* che inserisce il messaggio di errore associato all'errore corrente.

La funzione **vsyslog** è una variante di *syslog* che consente di formattare i messaggi di log utilizzando una lista di argomenti variabili (in stile *va\_list*).

```
#include <syslog.h>  
void vsyslog(int priority, const char *format, va_list arg);
```

## 11.4 Single-Instance Daemons

Alcuni demoni sono implementati in modo che solo una singola copia del demone alla volta dovrebbe essere in esecuzione per un corretto funzionamento. Un demone di questo tipo potrebbe, ad esempio, aver bisogno dell'accesso esclusivo a un dispositivo. Nel caso del demone cron, se fossero in esecuzione più istanze, ciascuna copia potrebbe tentare di avviare una singola operazione pianificata, risultando in operazioni duplicate e probabilmente in un errore.

### 11.4.1 Esempio 13.6

La seguente funzione illustra l'uso del blocco di file e record per garantire che sia in esecuzione solo una copia di un demone. Inizialmente viene definito con **LOCKFILE** il percorso del file di

---

```

#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <syslog.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>

#define LOCKFILE "/var/run/daemon.pid"
#define LOCKMODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

extern int lockfile(int);

int
already_running(void)
{
    int      fd;
    char    buf[16];

    fd = open(LOCKFILE, O_RDWR|O_CREAT, LOCKMODE);
    if (fd < 0) {
        syslog(LOG_ERR, "can't open %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    if (lockfile(fd) < 0) {
        if (errno == EACCES || errno == EAGAIN) {
            close(fd);
            return(1);
        }
        syslog(LOG_ERR, "can't lock %s: %s", LOCKFILE, strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf)+1);
    return(0);
}

```

---

lock utilizzato per identificare se il daemon è già in esecuzione e con LOCKMODE Modalità di permessi del file (rw-r-r-).

Nella funzione **already\_running** con la funzione open viene aperto il file di lock con permessi di lettura/scrittura. Dopo *lockfile(fd)* è una funzione esterna che applica un lock esclusivo sul file. Se il lock fallisce con EACCES o EAGAIN, significa che un altro processo ha già acquisito il lock, quindi chiude il file e restituisce 1. Per altri errori, registra l'errore con syslog e termina il programma. Con la funzione *ftruncate* pulisce il contenuto del file di lock, *getpid()* ottiene il PID del processo corrente e infine *write* scrive il PID nel file di lock, rendendo l'ID del processo disponibile per altri programmi o amministratori di sistema.

## 11.5 Daemon Conventions

Quando si implementa un daemon su sistemi Unix/Linux, è importante seguire alcune convenzioni standard per garantire che il daemon funzioni correttamente e sia compatibile con gli strumenti di sistema e le pratiche di amministrazione:

- se un daemon usa un file lock, il file si trova in /var/run. Il daemon dovrebbe avere i permessi del superuser per creare un file qui. Il nome del file è name.pid, dove name è il nome del daemon
- Se il daemon supporta opzioni di configurazione, queste si trovano in /etc. Il nome del file di configurazione è name.conf
- i daemons possono essere iniziate dalla linea di comando, ma generalmente iniziano da uno dei script di inizializzazione (/etc/tc\*or/etc/init.d/\*)
- Se un demone ha un file di configurazione, il demone legge il file all'avvio, ma di solito non lo guarderà più. Se un amministratore modifica la configurazione, il demone dovrebbe essere arrestato e riavviato per tenere conto dei cambiamenti configurazione. Per evitare ciò, alcuni demoni cattureranno SIGHUP e rileggeranno il loro file di configurazione quando ricevono il segnale. Dal momento che non sono associati con terminali, i demoni non hanno motivo di aspettare di ricevere SIGHUP.

### 11.5.1 Esempio 13.7

Il seguente programma mostra un modo per i daemon di leggere il suo file di configurazione.

---

```
#include "apue.h"
#include <pthread.h>
#include <syslog.h>

sigset_t    mask;
extern int already_running(void);

void
reread(void)
{
    /* ... */
}
```

```

void *
thr_fn(void *arg)
{
    int err, signo;

    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            syslog(LOG_ERR, "sigwait failed");
            exit(1);
        }

        switch (signo) {
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;

        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            exit(0);

        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }
    }
    return(0);
}

int
main(int argc, char *argv[])
{
    int                 err;
    pthread_t           tid;
    char                *cmd;
    struct sigaction    sa;

    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
    else
        cmd++;

    /*
     * Become a daemon.
     */
    daemonize(cmd);

    /*
     * Make sure only one copy of the daemon is running.
     */
    if (already_running()) {
        syslog(LOG_ERR, "daemon already running");
        exit(1);
    }

    /*
     * Restore SIGHUP default and block all signals.
     */
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) < 0)
        err_quit("%s: can't restore SIGHUP default");
}

```

```

    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "SIG_BLOCK error");

    /*
     * Create a thread to handle SIGHUP and SIGTERM.
     */
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0)
        err_exit(err, "can't create thread");

    /*
     * Proceed with the rest of the daemon.
     */
    /* ... */
    exit(0);
}

```

---

Tale funzione viene utilizzata per inizializzare un daemon. Quando tale funzione effettua il return, quindi termina, si può chiamare la funzione **already\_running** precedentemente vista per assicurarsi che solo una copia nel daemon è attiva. A questo punto SIGHUP viene ancora ignorato, quindi è necessario reimpostare la disposizione sul comportamento predefinito, in caso contrario, il thread che chiama sigwait potrebbe non vedere mai il segnale. Blocciamo tutti i segnali, come consigliato per i programmi multithread, e creiamo un file thread per gestire i segnali. L'unico compito del thread è attendere SIGHUP e SIGTERM. Quando riceve SIGHUP, il thread chiama reread per rileggere il suo file di configurazione. Quando riceve SIGTERM, il thread registra un messaggio ed esce.

# Capitolo 12

## Advanced I/O

### 12.1 Nonblocking I/O

In Linux, le operazioni di I/O bloccante (blocking I/O) sono quelle in cui il processo che effettua la richiesta viene sospeso fino a che l'operazione non è completata. Le cosiddette slow system calls sono ad esempio:

- Lettura da file: Quando si legge da un file con funzioni come `read()`, il processo si blocca fino a che i dati richiesti non sono completamente disponibili.
- Scrittura su file: Operazioni come `write()` bloccano il processo fino a quando i dati non sono stati effettivamente scritti sul disco.
- Operazioni su pipe o FIFO: La lettura da una pipe bloccante (`pipe()`) si ferma fino a che un altro processo non scrive dati.

L'I/O non bloccante è una tecnica usata in informatica per eseguire operazioni di input/output senza fermare l'esecuzione del programma mentre si attendono i risultati.

Esistono due modi per specificare il nonblocking I/O per un dato descrittore.

- Se chiamiamo `open` per ottenere il descrittore, possiamo specificare il flag `O_NONBLOCK`
- Per un descrittore già aperto, chiamiamo `fcntl` per attivare `O_NONBLOCK` status flag del file

### 12.1.1 Esempio 14.1 + 3.12

Il seguente programma in C legge fino a 500.000 byte dallo standard input (STDIN\_FILENO), scrive tali dati sullo standard output (STDOUT\_FILENO) in modalità non bloccante, e gestisce eventuali errori di scrittura. Alla fine, ripristina il file descriptor in modalità bloccante. Nel ciclo while continua a scrivere finché ci sono dati da scrivere (ntowrite > 0). Ogni chiamata a write tenta di scrivere i dati rimanenti (nwrite) a partire dal puntatore ptr. Gestione dei risultati di write:

- nwrite > 0: Avanzamento del puntatore e riduzione del conteggio ntowrite
- nwrite == -1 e errno == EAGAIN: Buffer pieno, riprova al successivo ciclo

Per creare un file di 500.000 per poter eseguire il comando facciamo:

```
dd if=/dev/urandom bs=1 count=3000000 | LC_ALL=C tr -dc [:alnum:] | head  
-c500000 > filegrosso
```

---

```
#include "apue.h"  
#include <errno.h>  
#include <fcntl.h>  
  
char    buf[500000];  
  
int  
main(void)  
{  
    int      ntowrite, nwrite;  
    char    *ptr;  
  
    ntowrite = read(STDIN_FILENO, buf, sizeof(buf));  
    fprintf(stderr, "read %d bytes\n", ntowrite);  
  
    set_f1(STDOUT_FILENO, O_NONBLOCK); /* set nonblocking */  
  
    ptr = buf;  
    while (ntowrite > 0) {  
        errno = 0;  
        nwrite = write(STDOUT_FILENO, ptr, ntowrite);  
        fprintf(stderr, "nwrite = %d, errno = %d\n", nwrite, errno);  
  
        if (nwrite > 0) {  
            ptr += nwrite;  
            ntowrite -= nwrite;  
        }  
    }  
  
    clr_f1(STDOUT_FILENO, O_NONBLOCK); /* clear nonblocking */  
  
    exit(0);  
}
```

---

## 12.2 Record Locking

Il record locking in Linux si riferisce a meccanismi che limitano l'accesso a specifiche parti di un file, consentendo a più processi di coordinare l'accesso a risorse condivise senza conflitti. Record Locking quindi è un termine utilizzato per descrivere l'abilità di un processo di impedire che altri processi possano modificare una parte di un file mentre il primo processo sta leggendo o modificando quella porzione di file. Il tipo di record blocking più utilizzato è il Blocco consultivo (*advisory locking*) in cui i processi devono cooperare per rispettare i blocchi, utilizzando apposite chiamate di sistema. Se un processo non verifica i blocchi, può ignorarli.

### 12.2.1 fcntl Record Locking

In Linux, il record locking viene gestito principalmente attraverso la funzione `fcntl()` che, già vista in precedenza, permette blocchi sia a livello di file intero che di porzioni (record) specifiche di un file:

```
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* struct flock *flockptr */ );
```

Restituisce in base a *cmd* in caso di successo e *-1* in caso di errore

Il parametro *fd* è il file descriptor del file aperto, quello *cmd* specifica il comando da eseguire tra:

- **F\_GETLK**: Determina se il lock descritto da *flockptr* è bloccato da qualche altro lock. Se esiste un blocco che impedisce la creazione del nostro, le informazioni su quel blocco esistente sovrascrivono le informazioni a cui punta *flockptr*.
- **F\_SETLK**: setta il blocco descritto da *flockptr*. Se stiamo cercando di ottenere un blocco in lettura (*l\_type* di **F\_RDLCK**) o un blocco in scrittura (*l\_type* di **F\_WRLCK**) e le regole di compatibilità non permettono al sistema di darci il blocco, *fcntl* restituisce *errno* settato a **EACCESS** o **EAGAIN**

- F\_SETLKW: questo comando è la versione bloccante di F\_SETLK con W nel comando che significa *wait*. Se la richiesta read lock o write lock non può essere soddisfatta perché un altro processo ha bloccato alcune parti della regione richiesta, il processo attende finché il blocco non è disponibile

Il terzo parametro *flockptr* è un puntatore ad una **struttura flock**:

```
struct flock {
    short l_type;    // Tipo di blocco: F_RDLCK, F_WRLCK, o F_UNLCK
    short l_whence; // Punto di partenza: SEEK_SET, SEEK_CUR, o SEEK_END
    off_t l_start;  // Offset rispetto a l_whence
    off_t l_len;    // Lunghezza del blocco (0 = bloccare fino alla fine del file)
    pid_t l_pid;    // PID del processo che detiene il blocco (per F_GETLK)
};
```

dove F\_RDLCK è il blocco di lettura, F\_WRLCK è il blocco di scrittura e F\_UNLCK rimuove il blocco.

### 12.2.2 Esempio lock2.c

Questo codice C implementa un semplice meccanismo di lock file basato sulla creazione di un file temporaneo (usando open con il flag O\_CREAT | O\_EXCL). Il comportamento del programma è il seguente:

Quando un processo tenta di aprire il file /tmp/LCK.test2 usando O\_CREAT | O\_EXCL, se il file non esiste, il processo lo crea e ottiene l'accesso esclusivo alla "regione critica" e garantisce che la creazione sia atomica, quindi se due processi provano a creare il file contemporaneamente, solo uno riesce. Se il file esiste già, la chiamata a open fallisce, segnalando che un altro processo detiene il "lock".

Se il file è creato con successo: il processo stampa un messaggio che conferma l'accesso esclusivo.

Rilascia il lock chiudendo il file descriptor (close) e rimuovendo il file (unlink).

Se il file esiste già: il processo stampa un messaggio che informa che il lock è già presente.

Aspetta 3 secondi prima di riprovare.

Ogni processo tenta di acquisire il lock fino a 10 volte. Dopo ogni tentativo fallito, il processo aspetta 3 secondi prima di riprovare.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

const char *lock_file = "/tmp/LCK.test2";

int main() {
    int file_desc;
    int tries = 10;

    while (tries--) {
        /*se il file viene creato da me va tutto bene, tuttavia se un altro tenta
        di creare lo stesso file che è già esistente, allora incontrerà dei problemi.
        Quindi l'altro utente se vuole creare lo stesso file lo deve prima rimuovere
        e lo farà dopo con unlink*/
        file_desc = open(lock_file, O_RDWR | O_CREAT | O_EXCL, 0444);
        if (file_desc == -1) {
            printf("%d - Lock already present\n", getpid());
            sleep(3);
        }
        else {
            /* critical region */
            printf("%d - I have exclusive access\n", getpid());
            sleep(1);
            (void)close(file_desc);
            (void)unlink(lock_file);
            /* non-critical region */
            sleep(2);
        }
    } /* while */
    exit(EXIT_SUCCESS);
}
```

### 12.2.3 Esempio lock3.c

Questo programma dimostra l'uso della funzione fcntl per impostare lock di file a livello di record in Linux:

Come nel codice precedente il file /tmp/test\_lock viene aperto (o creato se non esiste) con permessi di lettura e scrittura (O\_RDWR | O\_CREAT). Il file viene riempito con 100 caratteri 'A'. Questo garantisce che esista abbastanza spazio per applicare i lock.

A questo punto vengono definite le regioni da bloccare:

- **region\_1**: Blocca i byte da 10 a 30 con un blocco condiviso (shared lock, F\_RDLCK).

Questo tipo di blocco permette ad altri processi di leggere, ma non di scrivere in quella regione.

- **region\_2**: Blocca i byte da 40 a 50 con un blocco esclusivo (exclusive lock, F\_WRLCK).

Questo tipo di blocco impedisce ad altri processi di leggere o scrivere in quella regione.

Vengono impostati i lock con la **funzione fcntl** che viene usata con il comando F\_SETLK per impostare i lock definiti da region\_1 e region\_2. Se un lock non può essere acquisito (perché già bloccato da un altro processo), fcntl restituisce -1.

Il processo si mette in attesa per 60 secondi con sleep(60). Durante questo tempo, il file rimane bloccato, e altri processi che tentano di accedere alle regioni bloccate incontreranno un errore. Alla chiusura del file con close(file\_desc), i lock vengono automaticamente rilasciati.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";

int main() {
    int file_desc;
    int byte_count;
    char *byte_to_write = "A";
    struct flock region_1;
    struct flock region_2;
    int res;

    /* open a file descriptor */
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write\n", test_file);
        exit(EXIT_FAILURE);
    }

    /* put some data in the file */
    for (byte_count = 0; byte_count < 100; byte_count++) {
        (void)write(file_desc, byte_to_write, 1);
    }

    /* setup region 1, a shared lock, from bytes 10 -> 30 */
    region_1.l_type = F_RDLCK;
    region_1.l_whence = SEEK_SET;
    region_1.l_start = 10;
    region_1.l_len = 20;

    /* setup region 2, an exclusive lock, from bytes 40 -> 50 */
    region_2.l_type = F_WRLCK;
    region_2.l_whence = SEEK_SET;
    region_2.l_start = 40;
    region_2.l_len = 10;

    /* now lock the file */
    printf("Process %d locking file\n", getpid());
    res = fcntl(file_desc, F_SETLK, &region_1);
    if (res == -1) fprintf(stderr, "Failed to lock region 1\n");
    res = fcntl(file_desc, F_SETLK, &region_2);
    if (res == -1) fprintf(stderr, "Failed to lock region 2\n");

    /* and wait for a while */
    sleep(60);

    printf("Process %d closing file\n", getpid());
    close(file_desc);
    exit(EXIT_SUCCESS);
}
```

## 12.2.4 Esempio lock4.c

Questo programma C dimostra come utilizzare la funzione fcntl con il comando F\_GETLK per verificare la disponibilità di blocchi su un file:

Dopo l'apertura il file viene diviso in blocchi di 5 byte definiti dalla costante SIZE\_TO\_TRY.

Per ogni regione, il programma controlla se è possibile acquisire: un blocco esclusivo (write lock, F\_WRLCK) e un blocco condiviso (read lock, F\_RDLCK).

Uso di **F\_GETLK**: La funzione fcntl con il comando F\_GETLK restituisce informazioni sui lock esistenti che potrebbero impedire l'acquisizione di un nuovo lock sulla regione specificata.

Se non ci sono lock in conflitto, il campo l\_pid della struttura flock restituita sarà uguale a -1.

Per ogni regione testata, il programma stampa: se il blocco può essere acquisito (l\_pid == -1) oppure informazioni sui lock esistenti in caso di conflitto (usando la funzione show\_lock\_info)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";
#define SIZE_TO_TRY 5

void show_lock_info(struct flock *to_show);

int main() {
    int file_desc;
    int res;
    struct flock region_to_test;
    int start_byte;

    /* open a file descriptor */
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write", test_file);
        exit(EXIT_FAILURE);
    }

    for (start_byte = 0; start_byte < 99; start_byte += SIZE_TO_TRY) {
        /* set up the region we wish to test */
        region_to_test.l_type = F_WRLCK;
        region_to_test.l_whence = SEEK_SET;
        region_to_test.l_start = start_byte;
        region_to_test.l_len = SIZE_TO_TRY;
        region_to_test.l_pid = -1;

        printf("Testing F_WRLCK on region from %d to %d\n",
               start_byte, start_byte + SIZE_TO_TRY);
    }
}
```

```

        /* now test the lock on the file */
        res = fcntl(file_desc, F_GETLK, &region_to_test);
        if (res == -1) {
            fprintf(stderr, "F_GETLK failed\n");
            exit(EXIT_FAILURE);
        }
        if (region_to_test.l_pid != -1) {
            printf("Lock would fail. F_GETLK returned:\n");
            show_lock_info(&region_to_test);
        }
        else {
            printf("F_WRLCK - Lock would succeed\n");
        }

        /* now repeat the test with a shared (read) lock */

        /* set up the region we wish to test */
        region_to_test.l_type = F_RDLCK;
        region_to_test.l_whence = SEEK_SET;
        region_to_test.l_start = start_byte;
        region_to_test.l_len = SIZE_TO_TRY;
        region_to_test.l_pid = -1;

        printf("Testing F_RDLCK on region from %d to %d\n",
               start_byte, start_byte + SIZE_TO_TRY);

        /* now test the lock on the file */
        res = fcntl(file_desc, F_GETLK, &region_to_test);
        if (res == -1) {
            fprintf(stderr, "F_GETLK failed\n");
            exit(EXIT_FAILURE);
        }
        if (region_to_test.l_pid != -1) {
            printf("Lock would fail. F_GETLK returned:\n");
            show_lock_info(&region_to_test);
        }
        else {
            printf("F_RDLCK - Lock would succeed\n");
        }

    } /* for */

    close(file_desc);
    exit(EXIT_SUCCESS);
}

void show_lock_info(struct flock *to_show) {
    printf("\tl_type %d, ", to_show->l_type);
    printf("l_whence %d, ", to_show->l_whence);
    printf("l_start %d, ", (int)to_show->l_start);
    printf("l_len %d, ", (int)to_show->l_len);
    printf("l_pid %d\n", to_show->l_pid);
}

```

### 12.2.5 Esempio lock5.c

Questo programma C dimostra l'utilizzo di blocchi di file con la funzione fcntl. Riassumendo vengono applicati e rilasciati blocchi su diverse regioni del file: blocco condiviso (lettura) F\_RDLCK, blocco esclusivo (scrittura) F\_WRLCK e sblocco F\_UNLCK. Di seguito viene utilizzata la funzione fcntl:

- **F\_SETLK:** Tenta di acquisire o rilasciare un blocco senza attendere.
- **F\_SETLKW:** Tenta di acquisire un blocco, ma aspetta finché il blocco non diventa

disponibile

Quindi le operazioni che vengono svolte sono:

- Blocco condiviso (lettura): Regioni 10-15 e 40-50 vengono bloccate con F\_RDLCK. Consente ad altri processi di acquisire altri blocchi di lettura sulla stessa regione.
- Blocco esclusivo (scrittura): La regione 16-21 viene bloccata con F\_WRLCK. Esclude altri processi sia dalla lettura che dalla scrittura della regione.
- Sblocco: Le regioni 10-15 e 0-50 vengono sbloccate con F\_UNLCK.
- Blocco con attesa: La regione 16-21 viene bloccata nuovamente con F\_SETLK. Se la regione è bloccata da un altro processo, il programma aspetta.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>

const char *test_file = "/tmp/test_lock";

int main() {
    int file_desc;
    struct flock region_to_lock;
    int res;

    /* open a file descriptor */
    file_desc = open(test_file, O_RDWR | O_CREAT, 0666);
    if (!file_desc) {
        fprintf(stderr, "Unable to open %s for read/write\n", test_file);
        exit(EXIT_FAILURE);
    }

    region_to_lock.l_type = F_RDLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_RDLCK, region %d to %d\n",
           getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to lock region\n", getpid());
    } else {
        printf("Process %d - obtained lock region\n", getpid());
    }

    region_to_lock.l_type = F_UNLCK;
    region_to_lock.l_whence = SEEK_SET;
    region_to_lock.l_start = 10;
    region_to_lock.l_len = 5;
    printf("Process %d, trying F_UNLCK, region %d to %d\n",
           getpid(),
           (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
    res = fcntl(file_desc, F_SETLK, &region_to_lock);
    if (res == -1) {
        printf("Process %d - failed to unlock region\n", getpid());
    } else {
        printf("Process %d - unlocked region\n", getpid());
    }
}
```

```

region_to_lock.l_type = F_UNLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 0;
region_to_lock.l_len = 50;
printf("Process %d, trying F_UNLCK, region %d to %d\n", getpid(),
       (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
res = fcntl(file_desc, F_SETLK, &region_to_lock);
if (res == -1) {
    printf("Process %d - failed to unlock region\n", getpid());
} else {
    printf("Process %d - unlocked region\n", getpid());
}

region_to_lock.l_type = F_WRLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 16;
region_to_lock.l_len = 5;
printf("Process %d, trying F_WRLCK, region %d to %d\n", getpid(),
       (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
res = fcntl(file_desc, F_SETLK, &region_to_lock);
if (res == -1) {
    printf("Process %d - failed to lock region\n", getpid());
} else {
    printf("Process %d - obtained lock on region\n", getpid());
}

region_to_lock.l_type = F_RDLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 40;
region_to_lock.l_len = 10;
printf("Process %d, trying F_RDLCK, region %d to %d\n", getpid(),
       (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
res = fcntl(file_desc, F_SETLK, &region_to_lock);
if (res == -1) {
    printf("Process %d - failed to lock region\n", getpid());
} else {
    printf("Process %d - obtained lock on region\n", getpid());
}

region_to_lock.l_type = F_WRLCK;
region_to_lock.l_whence = SEEK_SET;
region_to_lock.l_start = 16;
region_to_lock.l_len = 5;
printf("Process %d, trying F_WRLCK with wait, region %d to %d\n", getpid(),
       (int)region_to_lock.l_start, (int)(region_to_lock.l_start + region_to_lock.l_len));
res = fcntl(file_desc, F_SETLKW, &region_to_lock);
if (res == -1) {
    printf("Process %d - failed to lock region\n", getpid());
} else {
    printf("Process %d - obtained lock on region\n", getpid());
}

printf("Process %d ending\n", getpid());
close(file_desc);
exit(EXIT_SUCCESS);
}

```

## 12.3 I/O Multiplexing

L'I/O Multiplexing è una tecnica di programmazione che consente a un singolo processo di monitorare più descrittori di file (file descriptors), come quelli associati a socket o file, per determinare quando sono pronti per essere letti o scritti. Questo permette di gestire più operazioni di I/O (Input/Output) in modo non bloccante, senza la necessità di utilizzare più thread o processi.

### 12.3.1 Funzione select()

La funzione `select()` consente di monitorare più descrittori di file per determinare se sono pronti per la lettura, scrittura o se ci sono errori. Mette il processo in stato di sleep per un tempo

massimo pari a timeout (tempo infinito se NULL) fino a quando almeno uno dei file descriptor appartenente a readfds, writefds o exceptfds non diviene attivo.

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Ritorna il numero di file descriptor pronti per l'I/O

Il parametro *nfds* indica il numero massimo di file descriptor da osservare incrementato di 1. Di seguito i parametri relativi ai set di file descriptor da monitorare: readfds (lettura), writefds (scrittura), exceptfds (errori).

I set di file descriptor possono contenere fino ad un massimo di FD\_SETSIZE file descriptor e di seguito le funzioni principali:

- Inizializza l'insieme vuoto: *voidFD\_ZERO(fd\_set \*set)*
- Inserisce il file descriptor *fd* nell'insieme: *voidFD\_SET(intfd, fd\_set \*set)*
- Rimuove il file descriptor *fd* dall'insieme: *voidFD\_CLR(intfd, fd\_set \*set)*
- Controlla se il file descriptor *fd* è nell'insieme: *intFD\_ISSET(intfd, fd\_set \*set)*

### Esempio Server echo con select

Nel seguente esempio il server si pone in ascolto di nuove connessioni da parte di client su un apposito socket, rileva la presenza di dati sui file descriptor attivi e risponde rispedendo al mittente lo stesso messaggio ricevuto.

#### 12.3.2 Funzione pselect()

La funzione pselect() in C è una variante di select() che aggiunge il supporto per i segnali. Come select(), viene utilizzata per monitorare più descrittori di file (file descriptors) per determinare se sono pronti per l'I/O (lettura, scrittura o errore). La differenza principale è che pselect() consente anche di specificare un set di segnali da bloccare durante l'attesa di eventi I/O.

```

#include <sys/select.h>
#include <signal.h>
int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            const struct timespec *timeout, const sigset_t *sigmask);

```

dove *sigmask* è una maschera di segnali da bloccare durante il monitoraggio dei file descriptor.

La funzione pselect() permette di eseguire atomicamente tre istruzioni:

- Imposta la maschera di segnali da bloccare con sigprocmask, salvando la maschera precedente
- Esegue la select
- Ripristina la precedente maschera

#### Esempio Server echo con pselect

### 12.3.3 Funzione poll()

La funzione poll() in C è un meccanismo di I/O multiplexing che consente di monitorare più descrittori di file (file descriptors) per determinare quando sono pronti per essere letti, scritti o per gestire errori. Come select(), poll() permette di gestire più operazioni di I/O in un solo thread o processo, senza dover bloccare l'esecuzione, ma con una sintassi e funzionalità leggermente più moderne.

```

#include <poll.h>
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

```

dove *fds* è un array di strutture pollfd e ogni struttura **pollfd** contiene:

```

struct pollfd {
    int fd;          // Il descrittore di file da monitorare
    short events;   // I tipi di eventi da monitorare
    short revents;  // I tipi di eventi che sono stati effettivamente rilevati
};

```

Il parametro *nfds* è il numero di descrittori di file da monitorare, ossia la dimensione dell'array *fds*; mentre il parametro *timeout* è il tempo massimo attenderà che uno dei descrittori di file diventi pronto, in particolare se è  $-1$  attenderà un tempo infinito, mentre se è  $0$  l'esecuzione e restituirà immediatamente.

### Esempio Server echo con poll

#### 12.3.4 Funzione epoll\_create() (solo LINUX)

La funzione `epoll_create()` è una delle funzioni principali per la gestione di I/O multiplexing su sistemi Linux. È utilizzata per creare un "epoll instance", che è una struttura interna per gestire efficientemente più descrittori di file.

```
#include <sys/epoll.h>
int epoll_create(int size);
```

La funzione `epoll_ctl()` è utilizzata per modificare il comportamento dell'istanza di epoll, come aggiungere, modificare o rimuovere descrittori di file da un'epoll instance.

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

Il parametro *epfd* è il descrittore di file che rappresenta l'istanza di epoll, ottenuto tramite la chiamata a `epoll_create()`. Il parametro *op* specifica l'operazione da eseguire:

- `EPOLL_CTL_ADD`: Aggiunge un nuovo descrittore di file all'epoll instance
- `EPOLL_CTL_MOD`: Modifica le proprietà di un descrittore di file già registrato
- `EPOLL_CTL_DEL`: Rimuove un descrittore di file dall'epoll instance

Infine il parametro *event* è un puntatore a una struttura `epoll_event`, che contiene le informazioni sugli eventi da monitorare o le modifiche sugli eventi di un descrittore di file esistente:

```

struct epoll_event {
    uint32_t events; // Tipi di eventi da monitorare
    union {
        void *ptr;
        int fd;
        uint32_t u32;
        uint64_t u64;
    } data; // Dati associati al descrittore di file
};
```

La funzione **epoll\_wait()** è utilizzata per attendere eventi su un'istanza di epoll e consente di bloccare l'esecuzione del programma fino a quando non si verificano eventi sui descrittori di file monitorati.

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

Il campo events della struttura epoll\_event descrive gli eventi che si sono verificati. Il campo data contiene le informazioni che erano state memorizzate prima dell'invocazione.

# Capitolo 13

## Interprocess Communication

La comunicazione interprocesso (Interprocess Communication, IPC) è un insieme di tecniche che consentono a processi separati di scambiare dati e informazioni. È fondamentale in sistemi multitasking o distribuiti, dove processi distinti devono collaborare per raggiungere obiettivi comuni. Ecco alcuni esempi:

Metodo	Velocità	Complessità	Sincronizzazione Necessaria	Relazione tra Processi	Scopo Tipico
Pipe	Medio	Bassa	No	Correlati	Flusso unidirezionale di dati
Socket	Medio	Media	No	Indipendenti	Comunicazione tra host/processi
Memoria Condivisa	Alta	Alta	Sì	Indipendenti	Accesso rapido a grandi quantità di dati
Semafori	N/A	Media	Sì	Indipendenti	Sincronizzazione
Message Queue	Medio	Media	No	Indipendenti	Comunicazione asincrona
Signal	Bassa	Bassa	No	Indipendenti	Notifica eventi
File Mappati	Alta	Media	Sì	Indipendenti	Condivisione di dati persistenti

### 13.1 Pipe

La Pipe è una tecnica di comunicazione tra due o più processi correlati tra di loro, come ad esempio tra il processo parent e il rispettivo child. La comunicazione può essere anche multili-

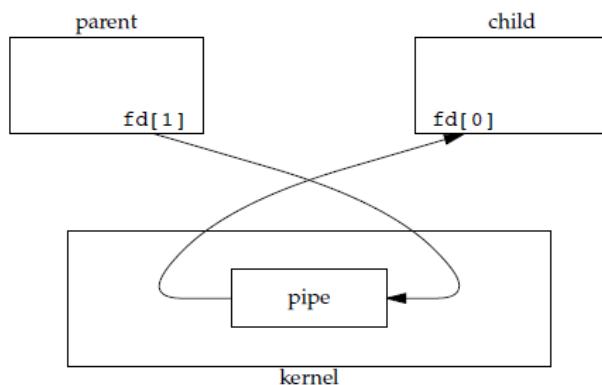
vello come la comunicazione tra processo parent, il child e il grand-child. La comunicazione è ottenuta mediante la scrittura di un processo nella pipe e la lettura dell'altro dalla pipe. Per ottenere la chiamata di sistema pipe, si creano due file, uno da scrivere nel file e un altro da leggere dal file.

Una pipe viene creata chiamando la funzione *pipe*:

```
#include <unistd.h>
int pipe(int fd[2]);
```

Due descrittori di file vengono restituiti tramite l'argomento *fd*: *fd[0]* è aperto in lettura e *fd[1]* è aperto per la scrittura. L'output di *fd[1]* è l'input per *fd[0]*. La funzione quindi crea una pipe unidirezionale tra due file descriptor e la lettura da *fd[0]* riceve dati scritti in *fd[1]*.

Normalmente, il processo che chiama *pipe* chiama poi *fork*, creando un canale IPC dal genitore al figlio o viceversa. Per una pipe dal genitore al figlio, il genitore chiude l'estremità di lettura della pipe (*fd[0]*) e il figlio chiude l'estremità di scrittura (*fd[1]*), quindi il parent scrive al child.



La tabella seguente rappresenta il comportamento di lettura dalle pipe o FIFO (First In, First Out) in base alla configurazione del flag *O\_NONBLOCK* e alle condizioni in cui vengono eseguite le operazioni di lettura. Ricordiamo che quando il flag *O\_NONBLOCK* è abilitato, le operazioni di I/O non bloccano il processo corrente in attesa di dati. In caso contrario, quando *O\_NONBLOCK* è disabilitato, il processo si blocca se non ci sono dati disponibili o se si verifica un errore (ad esempio, fine file, EOF). Specifichiamo che **p** rappresenta il numero di byte disponibili nel buffer della pipe o FIFO, mentre **n** è la quantità totale di dati che si cerca

di leggere.

O_NONBLOCK enabled?	Data bytes available in pipe or FIFO ( $p$ )			
	$p = 0$ , write end open	$p = 0$ , write end closed	$p < n$	$p \geq n$
No	block	return 0 (EOF)	read $p$ bytes	read $n$ bytes
Yes	fail (EAGAIN)	return 0 (EOF)	read $p$ bytes	read $n$ bytes

Quindi nel caso di **flag O\_NONBLOCK disabilitato**:

- Quando si legge da una pipe o FIFO e non ci sono dati disponibili, il processo si blocca fino a quando non viene ricevuto almeno un dato (block).
- Se la scrittura in una pipe o FIFO è chiusa ( $p = 0$ ), la lettura restituisce 0 (fine file, EOF).

Nel caso di **flag O\_NONBLOCK abilitato**:

- Se non ci sono dati disponibili, la lettura restituisce immediatamente un errore con EAGAIN (fail).
- Se ci sono meno dati di quanti richiesti, legge solo quei dati ( $p < n$ ).
- Se ci sono abbastanza dati, legge tutti i dati richiesti ( $p \geq n$ ).

La seconda tabella fornisce informazioni sui comportamenti del sistema durante l'operazione di scrittura su pipe o FIFO, sia quando il flag O\_NONBLOCK è abilitato sia disabilitato considerando PIPE\_BUF che rappresenta la dimensione massima del buffer per la comunicazione tra processi attraverso pipe o FIFO e la cui dimensione si può ottenere con *ulimit -a*:

O_NONBLOCK enabled?	$n \leq \text{PIPE\_BUF}$	$n > \text{PIPE\_BUF}$
No	Scrittura atomica di $n$ byte; potrebbe bloccarsi fino a quando non ci sono dati sufficienti per completare l'operazione.	Scrittura di $n$ byte; potrebbe bloccarsi fino a quando non sono disponibili dati sufficienti per completare l'operazione. Potrebbe esserci interleaving con le scritture di altri processi.
Si	Se c'è spazio sufficiente per scrivere immediatamente $n$ byte, allora <code>write()</code> riesce atomicamente; altrimenti fallisce con EAGAIN.	Se c'è spazio sufficiente per scrivere immediatamente alcuni byte, allora scrive tra 1 e $n$ byte; altrimenti fallisce con EAGAIN.

### 13.1.1 Esempio pipe2.c

Questo programma dimostra l'uso di una pipe per la comunicazione tra processi genitore e figlio in un ambiente Unix/Linux:

Viene chiamata la funzione pipe(file\_pipes) per creare una pipe, dove file\_pipes è un array di due file descriptor: file\_pipes[0] (Usato per leggere i dati dalla pipe) e file\_pipes[1] (Usato per scrivere i dati nella pipe).

fork() crea un nuovo processo figlio. Il valore di ritorno di fork() indica se il processo è il genitore o il figlio:

- Genitore (fork\_result > 0): Scrive nella pipe usando write(file\_pipes[1], some\_data, strlen(some\_data))
- Figlio (fork\_result == 0): Legge dalla pipe usando read(file\_pipes[0], buffer, BUFSIZ)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer));

    if (pipe(file_pipes) == -1) {
        fork_result = fork();
        if (fork_result == -1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
    }

    // We've made sure the fork worked, so if fork_result equals zero, we're in the child process.

    if (fork_result == 0) {
        data_processed = read(file_pipes[0], buffer, BUFSIZ);
        printf("Read %d bytes: %s\n", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }

    // otherwise, we must be the parent process.

    else {
        data_processed = write(file_pipes[1], some_data,
                               strlen(some_data));
        printf("Wrote %d bytes\n", data_processed);
    }
    exit(EXIT_SUCCESS);
}
```

### 13.1.2 Esempio pipe3.c

Questo programma dimostra l'uso di una pipe per la comunicazione tra processi e introduce l'esecuzione di un altro programma (pipe4) tramite la funzione execl nel processo figlio:

Il file descriptor file\_pipes[0] (lettura della pipe) viene convertito in una stringa e salvato in buffer usando sprintf.

Nel processo figlio viene chiamata la funzione execl per eseguire un altro programma chiamato pipe4. Argomenti passati a pipe4: pipe4 (Nome del programma da eseguire), buffer (Il valore di file\_pipes[0], convertito in stringa, che rappresenta il file descriptor della pipe) e (char \*)0 (Indica la fine della lista degli argomenti).

Nel processo parent scrive nella pipe il contenuto di some\_data ("123") tramite il file descriptor file\_pipes[1].

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    pid_t fork_result;

    memset(buffer, '\0', sizeof(buffer)); /*viene azzerato il buffer*/

    if (pipe(file_pipes) == -1) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }

        if (fork_result == 0) { /*child*/
            sprintf(buffer, "%d", file_pipes[0]);
            /*stampa con stringhe di formato e in file_pipes[0] vedremo il carattere ASCII 3*/
            (void)execl("pipe4", "pipe4", buffer, (char *)0); /*(char*)0 indica la fine della lista*/
            exit(EXIT_FAILURE);
        }
        else {
            data_processed = write(file_pipes[1], some_data,
                                   strlen(some_data));
            printf("%d - wrote %d bytes\n", getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}
```

### 13.1.3 Esempio pipe4.c

Il programma riportato è un esempio di utilizzo delle pipe in un contesto in cui il file descriptor viene passato come argomento dalla riga di comando:

Il programma accetta un argomento dalla riga di comando (argv[1]), che rappresenta un file

descriptor passato da un altro processo (ad esempio, dal programma che lo ha eseguito con `exec`). Usando il file descriptor convertito in un intero, il programma legge fino a `BUFSIZ` byte di dati dalla pipe.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int data_processed;
    char buffer[BUFSIZ + 1];
    int file_descriptor;

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &file_descriptor);
    data_processed = read(file_descriptor, buffer, BUFSIZ);

    printf("%d - read %d bytes: %s\n", getpid(), data_processed, buffer);
    exit(EXIT_SUCCESS);
}
```

### 13.1.4 Esempio pipe5.c

Questo programma dimostra una comunicazione tra processi utilizzando una pipe. Il processo genitore scrive nella pipe, e il processo figlio legge i dati dalla pipe e li passa al comando `od` (Octal Dump) per mostrarli in formato leggibile (opzione `-c`).

Nel **processo figlio**: viene chiuso il file descriptor 0 (standard input) e poi avviene la duplicazione di `file_pipes[0]`. Con `dup(file_pipes[0])`, il file descriptor più basso disponibile viene associato a `file_pipes[0]`, ovvero 0 (`stdin`). Il file descriptor duplicato (`file_pipes[0]`) viene chiuso, insieme al file descriptor di scrittura (`file_pipes[1]`), poiché non è necessario nel figlio. Con il comando `exec` viene lanciato il comando `od -c`, che legge dallo standard input e mostra i dati in formato leggibile. Lo standard input del comando è stato reindirizzato alla pipe.

Nel **processo parent**: Il genitore chiude il capo di lettura della pipe (`file_pipes[0]`). Scrive nella pipe (`file_pipes[1]`) la stringa `some_data ("123")`. Chiude il file descriptor di scrittura della pipe.

```

int main()
{
    int data_processed;
    int file_pipes[2];
    const char some_data[] = "123";
    pid_t fork_result;

    if (pipe(file_pipes) == 0) {
        fork_result = fork();
        if (fork_result == (pid_t)-1) {
            fprintf(stderr, "Fork failure");
            exit(EXIT_FAILURE);
        }
        /*Sappiamo che i file descriptor 0, 1, 2 sono sempre occupati, quindi quando si fa la fork vengono
        assegnati i numeri da 3 in poi. Nell'if seguente viene fatta una dup di 3 assegnandoli come file
        descriptor il numero intero più basso, quindi avendo chiuso prima la 0, verrà assegnato lo 0
        diventando così il capo di lettura della pipe*/
        if (fork_result == (pid_t)0) {
            close(0);
            dup(file_pipes[0]);
            close(file_pipes[0]);
            close(file_pipes[1]);

            execlp("od", "od", "-c", (char *)0);
            exit(EXIT_FAILURE);
        }
        else {
            close(file_pipes[0]);
            data_processed = write(file_pipes[1], some_data,
                                  strlen(some_data));
            close(file_pipes[1]);
            printf("%d - wrote %d bytes\n", (int)getpid(), data_processed);
        }
    }
    exit(EXIT_SUCCESS);
}

```

### 13.1.5 Esempio pipe\_ls\_wc.c

Questo programma implementa un esempio classico di comunicazione tra processi tramite pipe, in cui due processi figli cooperano per eseguire il comando `ls | wc -l`. Il primo figlio esegue il comando `ls` e invia la sua output attraverso una pipe al secondo figlio, che esegue il comando `wc -l` per contare le righe.

**Primo processo figlio (ls):** Chiude l'estremità di lettura della pipe (`pf[0]`), poiché il processo figlio non ne ha bisogno. Duplica il file descriptor di scrittura della pipe (`pf[1]`) nello standard output (`STDOUT_FILENO`), assicurandosi che tutto ciò che `ls` scrive vada nella pipe. Chiude `pf[1]` dopo la duplicazione. Esegue `ls`, che scrive il suo output nella pipe.

**Secondo processo figlio (wc -l):** Chiude l'estremità di scrittura della pipe (`pf[1]`), poiché il processo figlio non ne ha bisogno. Duplica il file descriptor di lettura della pipe (`pf[0]`) nello standard input (`STDIN_FILENO`), in modo che `wc -l` legga dalla pipe. Chiude `pf[0]` dopo la duplicazione. Esegue `wc -l`, che legge il numero di righe dall'input e le stampa.

**Processo genitore:** Chiude entrambi gli estremi della pipe, poiché non sono più necessari. Aspetta che entrambi i figli terminino (`wait`).

```

int
main(int argc, char *argv[])
{
    int pfd[2];                                /* Pipe file descriptors */

    if (pipe(pfd) == -1)                         /* Create pipe */
        errExit("pipe");

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:           /* First child: exec 'ls' to write to pipe */
        if (close(pfd[0]) == -1)                  /* Read end is unused */
            errExit("close 1");

        /* Duplicate stdout on write end of pipe; close duplicated descriptor */

        if (pfd[1] != STDOUT_FILENO) {             /* Defensive check */
            if (dup2(pfd[1], STDOUT_FILENO) == -1)
                errExit("dup2 1");
            if (close(pfd[1]) == -1)
                errExit("close 2");
        }

        execlp("ls", "ls", (char *) NULL);        /* Writes to pipe */
        errExit("execlp ls");

    default:          /* Parent falls through to create next child */
        break;
    }

    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:           /* Second child: exec 'wc' to read from pipe */
        if (close(pfd[1]) == -1)                  /* Write end is unused */
            errExit("close 3");

        /* Duplicate stdin on read end of pipe; close duplicated descriptor */

        if (pfd[0] != STDIN_FILENO) {             /* Defensive check */
            if (dup2(pfd[0], STDIN_FILENO) == -1)
                errExit("dup2 2");
            if (close(pfd[0]) == -1)
                errExit("close 4");
        }

        execlp("wc", "wc", "-l", (char *) NULL);
        errExit("execlp wc");

    default: /* Parent falls through */
        break;
    }

    /* Parent closes unused file descriptors for pipe, and waits for children */

    if (close(pfd[0]) == -1)
        errExit("close 5");
    if (close(pfd[1]) == -1)
        errExit("close 6");
    if (wait(NULL) == -1)
        errExit("wait 1");
    if (wait(NULL) == -1)
        errExit("wait 2");

    exit(EXIT_SUCCESS);
}

```

## 13.2 FIFOs

FIFO (First In, First Out), noto anche come FIFO file, è un tipo speciale di file utilizzato per la comunicazione interprocesso, simile alle pipe, ma può essere utilizzato tra processi non correlati (ad esempio, tra processi avviati indipendentemente). I file FIFO consentono a due processi separati di comunicare tra loro, uno scrivendo dati e l'altro leggendo quei dati in ordine sequenziale (il primo dato scritto è il primo ad essere letto). A differenza delle pipe anonime, un file FIFO esiste nel file system come un file speciale, visibile e accessibile tramite il suo nome. Creare un FIFO è molto simile a creare un file, infatti il pathname di un FIFO esiste nel filesystem:

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
int mkfifoat(int dirfd, const char *path, mode_t mode);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove con *mode* vengono specificati i permessi come nel caso della funzione open.

La funzione mkfifoat è una variante della funzione mkfifo, introdotta per fornire maggiore flessibilità nel creare un file FIFO in relazione a un descrittore di directory:

- se *path* è una percorso assoluto, il paramtro *fd* viene ignorato e la funzione mkfifoat funziona come mkfifo
- se *path* è un percorso relativo e *fd* è un file descriptor valido di una dir aperta, il percorso sarà considerato relativo rispetto a tale directory
- se *path* è un percorso relativo e *fd* ha il valore speciale AT\_FDCWD, il percorso sarà considerato relativo alla current working directory

Dopo la creazione della FIFO, possiamo aprirla utilizzando la **funzione open**:

- senza il flag O\_NONBLOCK: un'apertura per sola lettura blocca finché qualche altro processo non apre la FIFO per la scrittura. Allo stesso modo, un open per sola scrittura blocca finché qualche altro processo non apre la FIFO per la lettura.

- con il flag O\_NONBLOCK: una open in sola lettura ritorna immediatamente, ma una open in sola scrittura ritorna errore se nessun processo ha la FIFO aperta per lettura. Se viene scritto su una FIFO che nessun processo ha aperto per la lettura, viene generato il segnale SIGPIPE.

### 13.3 XSI IPC

Per permettere la comunicazione di due processi non imparentati tra di loro, bisogna trovare un punto d'incontro, ovvero una chiave che entrambi i processi devono avere. Per questo si parla di XSI IPC, ovvero una serie di interfacce per la comunicazione tra processi che comprendono:

- Shared Memory
- Message Queues
- Semaphores

Ogni struttura IPC (coda di messaggi, semaforo o segmento di memoria condivisa) nel file kernel è indicato da un identificatore intero non negativo. Per inviare un messaggio o recuperare un messaggio da una coda di messaggi tutto ciò che ci serve sapere è l'identificatore del file coda. A differenza dei descrittori di file, gli identificatori IPC non sono numeri interi piccoli.

La funzione `ftok()` (File Key) è una funzione di sistema utilizzata in ambito Unix/Linux per generare una chiave univoca (`key`) da un file specifico o da un percorso.

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

L'argomento `path` deve fare riferimento a un file esistente e poi vengono utilizzati solo gli 8 bit inferiori dell'ID del progetto durante la generazione della chiave. Quindi si potrebbe fare la chiamata `ftok()` con lo stesso `path` più volte assegnando ogni volta un `id` diverso.

Il comando `ipcs` mostra tutte le informazioni riguardanti gli IPC system file, quindi ci permette di ottenere l'elenco di shared memory segments, message queues, e semaphore arrays.

## 13.4 Semaphores

I semafori sono strumenti di sincronizzazione utilizzati nei sistemi operativi e nella programmazione concorrente per gestire l'accesso a risorse condivise da parte di più processi o thread. Servono a evitare conflitti e garantire un utilizzo coordinato delle risorse, prevenendo problemi come le condizioni di gara (race condition) o l'accesso non sicuro ai dati.

Un semaforo è essenzialmente una variabile intera che tiene traccia di una risorsa disponibile o del numero di risorse disponibili. Le operazioni principali che si possono effettuare su un semaforo sono:

- Wait (P, "prova a prendere la risorsa"): Se il valore del semaforo è maggiore di 0, lo decrementa e permette al processo/thread di continuare. Se il valore è 0, il processo/thread si blocca e attende fino a quando il semaforo non viene incrementato.
- Signal (V, "rilascia la risorsa"): Incrementa il valore del semaforo e, se ci sono processi/thread in attesa, ne sblocca uno.

Esistono due tipi di semaforo:

- Semaforo Binario: Può assumere solo i valori 0 o 1. Simile a un mutex (mutual exclusion), viene utilizzato per garantire che una risorsa venga utilizzata da un solo processo o thread alla volta.
- Semaforo Contatore: Può assumere qualsiasi valore intero non negativo. Viene utilizzato per gestire l'accesso a risorse con un numero limitato di istanze (ad esempio, un buffer con 5 spazi disponibili).

Un semaforo, in molti sistemi (specialmente in Unix/Linux), non è semplicemente un singolo valore non negativo, ma un insieme di valori di semaforo, dove ciascun valore nell'insieme può essere utilizzato per rappresentare risorse o condizioni differenti. Nel kernel ogni set di semafori ha una struttura **semid\_ds**.

Quando vogliamo utilizzare un semaforo, bisogna prima ottenere l'ID del set di semafori chiamando la funzione **semget**:

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

Restituisce l'ID del semaforo in caso di successo e  $-1$  in caso di errore

Quando creiamo un semaforo utilizzando la funzione `semget`, possiamo specificare il numero di valori (o elementi) nel set di semafori tramite il parametro `nsems`. Ogni elemento del set può essere gestito separatamente per sincronizzare l'accesso a risorse multiple o per rappresentare condizioni diverse.

I flag comuni sono `IPC_CREAT` (per creare l'array, altrimenti ne stai appena aprendo uno esistente) e `IPC_EXCL` se vuoi che la chiamata fallisca quando esiste già.

La funzione `semctl` consente di leggere o modificare i valori dei semafori, ottenere informazioni sul set di semafori e gestire operazioni amministrative, come l'eliminazione di un set:

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

dove `semid` è l'identificatore del set di semafori, ottenuto tramite la funzione `semget`, `semnum` specifica quale semaforo del set verrà manipolato, `cmd` indica il comando che vogliamo eseguire e `arg` che è un parametro opzionale di tipo `semun` e fornisce dati addizionali necessari per alcuni comandi (ad esempio, impostare un valore o modificare una struttura):

```
union semun {
    int val;           // Per SETVAL
    struct semid_ds *buf; // Per IPC_STAT e IPC_SET
    unsigned short *array; // Per GETALL e SETALL
};
```

La funzione `semop` è utilizzata nei sistemi Unix/Linux per eseguire operazioni atomiche sui semafori di un set. Questa funzione consente di incrementare, decrementare o aspettare un semaforo (o un gruppo di semafori) in modo sincrono e garantisce che le operazioni siano eseguite senza interferenze da parte di altri processi.

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove *nops* indica il numero delle operazioni indicate nel vettore *semoparray* che è un puntatore ad un array di operazioni da eseguire su uno specifico semaforo e rappresentate da una struttura **sembuf**:

```
struct sembuf {
    unsigned short sem_num; // Indice del semaforo nel set
    short sem_op;           // Operazione sul semaforo (negativo, 0, positivo)
    short sem_flg;          // Flag di controllo
};
```

Poichè nella maggior parte dei casi abbiamo bisogno di un semplice semaforo singolo e binario, le chiamate da effettuare sono:

- **Creare la chiave:**

```
akey = ftok("/home/yourhome/your_dir/your_file",1);
```

- **Creare il semaforo:**

```
asem = semget(akey, 1, IPC_CREAT | IPC_EXCL);
```

- **Inizializzare il semaforo:**

```
sem_union.val = 1;
semctl(asem, 0, SETVAL, sem_union);
```

- **Preparare l'operazione:**

```

struct sembuf sem_b;
sem_b.sem_num = 0;
sem_b.sem_op = -1; /* P() */
or
sem_b.sem_op = 1; /* V() */
sem_b.sem_flg = SEM_UNDO;

```

- Effettuare l'operazione:

```
semop(asem, &sem_b, 1);
```

- Rimuovere il semaforo:

```
semctl(asem, 0, IPC_RMID, sem_union);
```

I semafori si possono classificare in due principali categorie: named semaphores (semafori nominati) e unnamed semaphores (semafori anonimi). Entrambi i tipi sono utilizzati per la sincronizzazione tra processi, ma differiscono nel modo in cui vengono creati, identificati e utilizzati.

Caratteristica	Named Semaphores	Unnamed Semaphores
Identificazione	Nome globale nel file system	Puntatore in memoria
Ambito	Processi non correlati	Processi o thread correlati
Persistenza	Persistono dopo la terminazione del processo (finché non vengono rimossi)	Esistono solo durante la vita dei processi/thread
Creazione	<code>sem_open</code> , <code>sem_unlink</code>	<code>sem_init</code> , <code>sem_destroy</code>

I **named semaphores** sono identificati da un nome unico nel file system, simile a un file e possono essere utilizzati per sincronizzare processi non correlati, poiché il nome è accessibile globalmente. Le funzioni per aprire un semaforo (il primo) o per crearne uno nuovo (il secondo):

```

sem_t *sem_open(const char *, int oflag, mode_t mode, unsigned int value);
sem_t *sem_open(const char *, int oflag);

```

I flag comuni sono O\_CREAT (per creare un semaforo) o O\_EXCL se vuoi che la chiamata fallisca quando esiste già. Il terzo e il quarto parametro sono necessari solo quando si sta creando un nuovo semaforo e specificano i privilegi e il valore iniziale del semaforo.

Per rimuovere un semaforo:

```
int sem_unlink(sem_t *);
```

Gli **unnamed semaphores** possono essere utilizzati solo per sincronizzare processi correlati (ad esempio, processi padre e figlio) o thread all'interno dello stesso processo. Per creare e distruggere un semaforo sono utilizzate le seguenti funzioni:

```

#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);

```

dove se pshared è 0 il semaforo è condiviso tra thread dello stesso processo, mentre se è 1 il semaforo è condiviso tra processi diversi (richiede memoria condivisa).

### 13.4.1 Esempio psem\_create.c

Il seguente codice è un esempio di programma che utilizza i named semaphores. Questo programma consente di creare o aprire un semaforo nominato (named semaphore) con opzioni specifiche passate tramite riga di comando tenendo conto che argc indica il numero degli argomenti, mentre argv sono effettivamente gli argomenti. Il ciclo while con getopt analizza gli argomenti della riga di comando per determinare le opzioni da applicare:

- -c: Se specificato, il programma aggiunge il flag O\_CREAT che crea un nuovo semaforo se non esiste.
- -x: Se specificato, aggiunge il flag O\_EXCL, che causa un errore se il semaforo esiste già (usato insieme a O\_CREAT).

Se non vengono fornite opzioni valide, la funzione usageError(argv[0]) viene chiamata per mostrare l'uso corretto del programma.

Il nome del semaforo è obbligatorio e deve essere specificato come argomento (indicato da argv[optind]). Se non viene passato un nome, il programma segnala un errore.

Se il programma riceve ulteriori argomenti, essi vengono utilizzati per configurare i permessi e il valore iniziale del semaforo:

- **Permessi (perms):** Se non specificati, i permessi predefiniti sono lettura e scrittura per l'utente, ottenuti combinando le costanti S\_IRUSR e S\_IWUSR. Se specificati, i permessi vengono interpretati come un numero ottale (getInt(argv[optind + 1], GN\_BASE\_8, ...)).
- **Valore Iniziale (value):** Se non specificato, il valore predefinito è 0. Se specificato, viene interpretato come un numero intero (getInt(argv[optind + 2], 0, ...)).

La funzione sem\_open viene utilizzata per creare o aprire un semaforo nominato con i parametri specificati.

```

static void
usageError(const char *progName)
{
    fprintf(stderr, "Usage: %s [-cx] name [octal-perms [value]]\n", progName);
    fprintf(stderr, "      -c   Create semaphore (O_CREAT)\n");
    fprintf(stderr, "      -x   Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int
main(int argc, char *argv[])
{
    int flags, opt;
    mode_t perms;
    unsigned int value;
    sem_t *sem;

    flags = 0;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
            case 'c':   flags |= O_CREAT;           break;
            case 'x':   flags |= O_EXCL;           break;
            default:   usageError(argv[0]);
        }
    }

    if (optind >= argc)
        usageError(argv[0]);

    /* Default permissions are rw-----; default semaphore initialization
     * value is 0 */

    perms = (argc <= optind + 1) ? (S_IRUSR | S_IWUSR) :
        getInt(argv[optind + 1], GN_BASE_8, "octal-perms");
    value = (argc <= optind + 2) ? 0 : getInt(argv[optind + 2], 0, "value");

    sem = sem_open(argv[optind], flags, perms, value);
    if (sem == SEM_FAILED)
        errExit("sem_open");

    exit(EXIT_SUCCESS);
}

```

Creazione con permessi e valore personalizzato:

```
./psem_create -c /mysem 0644 5
```

### 13.4.2 Esempio sem1.c

Il codice fornito è un esempio di utilizzo di semafori SysV per sincronizzare l'accesso a una sezione critica.

Nella funzione **main** vengono effettuate le seguenti operazioni:

- **Inizializzazione del semaforo (semget):** Il programma utilizza la funzione semget per creare un semaforo con una chiave specifica (1234) e i permessi 0666 (lettura e scrittura per tutti gli utenti).
- **Inizializzazione del semaforo (set\_semvalue):** Se viene passato un argomento da riga di comando, il programma chiama la funzione set\_semvalue per impostare il valore iniziale del semaforo. Se l'inizializzazione fallisce, il programma esce con un errore.
- **Carattere identificativo:** Il programma usa due caratteri ('O' e 'X') per distinguere i processi che accedono alla sezione critica:
  - 'O': Processo senza argomenti da riga di comando.
  - 'X': Processo con argomenti da riga di comando.
- **Ciclo Principale:** Il programma entra ed esce dalla sezione critica per 10 iterazioni:
  - Richiesta di accesso alla sezione critica: Usa semaphore\_p per decrementare il semaforo (operazione P, o wait).
  - Accesso alla sezione critica: Stampa un carattere (op\_char), attende un tempo casuale e stampa nuovamente il carattere.
  - Rilascio della sezione critica: Usa semaphore\_v per incrementare il semaforo (operazione V, o signal).
  - Attesa casuale: Attende un intervallo di tempo casuale prima della successiva iterazione.

- **Pulizia:** Se il programma viene eseguito con un argomento da riga di comando, chiama la funzione `del_semvalue` per rimuovere il semaforo.

```

static int sem_id;

int main(int argc, char *argv[])
{
    int i;
    int pause_time;
    char op_char = 'O';
    /*inizializza il generatore di numeri pseudo-casuali e in questo modo imposta il seed*/
    srand((unsigned int)getpid());
    /*Creazione del semaforo con i privilegi 0666*/
    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);

    if (argc > 1) {
        if (!set_semvalue()) {
            fprintf(stderr, "Failed to initialize semaphore\n");
            exit(EXIT_FAILURE);
        }
        /*Quindi se c'è l'argomento vogliamo lavorare sul char 'X', altrimenti
        avremmo lavorato sul char 'O'*/
        op_char = 'X';
        sleep(2);
    }

    /* Then we have a loop which enters and leaves the critical section ten times.
    There, we first make a call to semaphore_p which sets the semaphore to wait, as
    this program is about to enter the critical section. */

    for(i = 0; i < 10; i++) {

        if (!semaphore_p()) exit(EXIT_FAILURE);
        printf("%c", op_char);fflush(stdout);
        pause_time = rand() % 3;
        /*Sleep casuale*/
        sleep(pause_time);
        printf("%c", op_char);fflush(stdout);

        /* After the critical section, we call semaphore_v, setting the semaphore available,
        before going through the for loop again after a random wait. After the loop, the call
        to del_semvalue is made to clean up the code. */

        if (!semaphore_v()) exit(EXIT_FAILURE);

        pause_time = rand() % 2;
        sleep(pause_time);
    }

    printf("\n%d - finished\n", getpid());

    if (argc > 1) {
        sleep(10);
        del_semvalue();
    }

    exit(EXIT_SUCCESS);
}

/* The function set_semvalue initializes the semaphore using the SETVAL command in a
semctl call. We need to do this before we can use the semaphore. */

static int set_semvalue(void)
{
    union semun sem_union;

    sem_union.val = 1;
    /*SETVAL è il comando per inizializzare*/
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
    return(1);
}

```

```

/* The del_semvalue function has almost the same form, except the call to semctl uses
   the command IPC_RMID to remove the semaphore's ID. */

static void del_semvalue(void)
{
    union semun sem_union;

    if (semctl(sem_id, 0, IPC_RMID /*, sem_union*/) == -1)
        fprintf(stderr, "Failed to delete semaphore\n");
}

/* semaphore_p changes the semaphore by -1 (waiting). */

static int semaphore_p(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = -1; /* P() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return(0);
    }
    return(1);
}

/* semaphore_v is similar except for setting the sem_op part of the sembuf structure to 1,
   so that the semaphore becomes available. */

static int semaphore_v(void)
{
    struct sembuf sem_b;

    sem_b.sem_num = 0;
    sem_b.sem_op = 1; /* V() */
    sem_b.sem_flg = SEM_UNDO;
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return(0);
    }
    return(1);
}

```

## 13.5 Shared Memory

La shared memory (memoria condivisa) è una forma di comunicazione interprocesso (IPC) che consente a più processi di accedere a un'area comune di memoria per scambiarsi dati in modo rapido ed efficiente. È uno dei metodi più veloci per la comunicazione tra processi, poiché evita il passaggio di messaggi o dati tramite il sistema operativo una volta che la memoria è stata mappata.

Un processo crea un'area di memoria condivisa utilizzando il sistema di IPC, tipicamente tramite la funzione **shmget** che restituisce un ID del segmento:

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int flag);
```

Restituisce l'ID della memoria condivisa in caso di successo o  $-1$  in caso di errore

dove *size* è la dimensione in byte della memoria condivisa e *flag* per permessi (es., 0666) e opzioni (IPC\_CREAT, IPC\_EXCL).

I processi accedono alla memoria condivisa mappandola nel loro spazio di indirizzi con la funzione **shmat**, potendo così leggere o scrivere direttamente nella memoria.

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

Restituisce il puntatore alla memoria condivisa o -1 in caso di errore

I processi scrivono e leggono i dati direttamente dalla memoria condivisa con la funzione **shmctl**:

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf );
```

Restituisce 0 in caso di successo e -1 in caso di errore

I processi dissociano la memoria condivisa dal loro spazio di indirizzi con **shmdt**:

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove *addr* è il valore restituito dalla funzione *shmdt*

### 13.5.1 Esempio shm1.c

Il seguente codice è un esempio di un programma che utilizza la memoria condivisa in un sistema Unix-like, per la comunicazione tra due processi. Qui vediamo un **consumatore che legge i dati dalla memoria condivisa** e stampa ciò che è stato scritto da un altro processo (un produttore). Il programma utilizza la memoria condivisa per scambiare dati tra processi in modo sicuro e sincronizzato.

Il programma comincia con l'inizializzazione della memoria condivisa. Viene creato un segmento di memoria condivisa con l'ID shmid utilizzando la funzione shmget. Se la creazione della memoria fallisce, viene stampato un errore e il programma termina.

La funzione shmat mappa il segmento di memoria condivisa nell'indirizzo virtuale del processo. Restituisce un puntatore alla memoria condivisa, che può essere usato per leggere e scrivere i dati. Se l'operazione fallisce, restituisce (void \*)-1.

Il programma usa la struttura shared\_use\_st per leggere e scrivere i dati nella memoria condivisa. La struttura shared\_use\_st non è definita nel codice che hai fornito, ma presumibilmente contiene almeno due membri:

- some\_text: Una stringa contenente il testo scritto dal produttore.
- written\_by\_you: Un flag che indica se il produttore ha scritto qualcosa nella memoria condivisa

Il consumatore aspetta un tempo casuale tra 0 e 3 secondi. Se il testo scritto è "end", il consumatore esce dal ciclo e termina.

Quando il processo consumatore ha finito di leggere dalla memoria condivisa, deve dissociarsi dal segmento di memoria con la funzione shmdt e poi rimuoverlo con shmctl.

```
int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    int shmid;

    srand((unsigned int) getpid());
    /* La size è di 2052 byte, ovvero 2048+4 poichè è char*/
    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }

    /* We now make the shared memory accessible to the program. */

    shared_memory = shmat(shmid, (void *)0, 0);
    if (shared_memory == (void *)-1) {
        fprintf(stderr, "shmat failed\n");
        exit(EXIT_FAILURE);
    }

    printf("Memory attached at %X\n", (int)shared_memory);
    /* The next portion of the program assigns the shared_memory segment to shared_stuff,
       which then prints out any text in written_by_you. The loop continues until end is found
       in written_by_you. The call to sleep forces the consumer to sit in its critical section,
       which makes the producer wait. */
}
```

```

shared_stuff = (struct shared_use_st *)shared_memory;
shared_stuff->written_by_you = 0;
while(running) {
    if (shared_stuff->written_by_you) {
        printf("You wrote: %s", shared_stuff->some_text);
        sleep( rand() % 4 ); /* make the other process wait for us ! */
        shared_stuff->written_by_you = 0;
        if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
            running = 0;
        }
    }
}

/* Lastly, the shared memory is detached and then deleted. */

if (shmctl(shmid, IPC_RMID, 0) == -1) {
    fprintf(stderr, "shmctl failed\n");
    exit(EXIT_FAILURE);
}

if (shmctl(shmid, IPC_RMID, 0) == -1) {
    fprintf(stderr, "shmctl(IPC_RMID) failed\n");
    exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}

```

### 13.5.2 Esempio shm2.c

Il codice è un esempio di un produttore che utilizza memoria condivisa per comunicare con un altro processo (il consumatore). In questo scenario, il **produttore scrive dei dati nella memoria condivisa** che il consumatore può leggere.

Il ciclo while principale del programma consente al produttore di scrivere dati nella memoria condivisa finché non viene inserito il comando end. Il produttore attende finché il consumatore non ha letto i dati. Questo è controllato dalla variabile written\_by\_you. Se il valore è 1, significa che i dati non sono ancora stati letti. Il programma accetta un input dall'utente utilizzando fgets. Il testo inserito viene copiato nella memoria condivisa utilizzando strncpy. La lunghezza massima è definita da TEXT\_SZ. Il flag written\_by\_you viene impostato a 1 per indicare al consumatore che ci sono nuovi dati disponibili.

```

int main()
{
    int running = 1;
    void *shared_memory = (void *)0;
    struct shared_use_st *shared_stuff;
    char buffer[BUFSIZ];
    int shmid;

    shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

    if (shmid == -1) {
        fprintf(stderr, "shmget failed\n");
        exit(EXIT_FAILURE);
    }
}

```

```

shared_memory = shmat(shmid, (void *)0, 0);
if (shared_memory == (void *)-1) {
    fprintf(stderr, "shmat failed\n");
    exit(EXIT_FAILURE);
}

printf("Memory attached at %X\n", (int)shared_memory);

shared_stuff = (struct shared_use_st *)shared_memory;
while(running) {
    while(shared_stuff->written_by_you == 1) {
        sleep(1);
        printf("waiting for client...\n");
    }
    printf("Enter some text: ");
    fgets(buffer, BUFSIZ, stdin);

    strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
    shared_stuff->written_by_you = 1;

    if (strncmp(buffer, "end", 3) == 0) {
        running = 0;
    }
}

if (shmrdt(shared_memory) == -1) {
    fprintf(stderr, "shmrdt failed\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}

```

# Capitolo 14

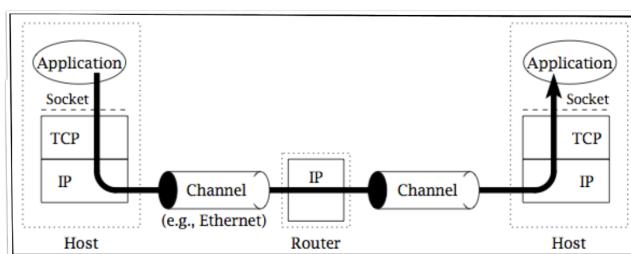
## UNIX Network Programming

### 14.1 Sockets Programming

Il **Socket Programming** è un metodo usato in programmazione per consentire la comunicazione tra due dispositivi su una rete, come un computer client e un server, oppure sullo stesso host. I socket sono punti di comunicazione che consentono di inviare e ricevere dati tra due nodi di una rete.

Un **socket** è un’interfaccia software che permette a un’applicazione di interagire con la rete. È come un’estremità di un tubo attraverso cui passano i dati tra due processi su una rete. I socket possono essere di due tipi:

- **Socket TCP (Stream Socket)**: Garantisce una connessione affidabile tra client e server. Utilizza il protocollo TCP (Transmission Control Protocol).
- **Socket UDP (Datagram Socket)**: Non garantisce la consegna dei pacchetti. Utilizza il protocollo UDP (User Datagram Protocol), più veloce ma meno affidabile.



Il **TCP/IP (Transmission Control Protocol/Internet Protocol)** è una suite di protocolli di comunicazione che costituisce la base di Internet e della maggior parte delle reti private. Consente ai dispositivi di connettersi e scambiare dati in modo standardizzato. La suite TCP/IP si basa su un'architettura a quattro livelli, ognuno con uno scopo specifico:

- **Livello Applicativo:** Fornisce le interfacce che le applicazioni utilizzano per accedere ai servizi di rete. Esempi: HTTP (navigazione web), FTP (trasferimento file), SMTP (email), DNS (risoluzione dei nomi di dominio).
- **Livello di Trasporto:** Gestisce la comunicazione end-to-end tra i dispositivi, garantendo l'affidabilità e l'integrità dei dati.

Protocolli principali:

- TCP: Connessione affidabile, orientata alla connessione.
- UDP: Connessione non affidabile, più veloce, orientata ai datagrammi.

- **Livello Internet:** Gestisce l'instradamento dei pacchetti tra i dispositivi in diverse reti.

Protocollo principale:

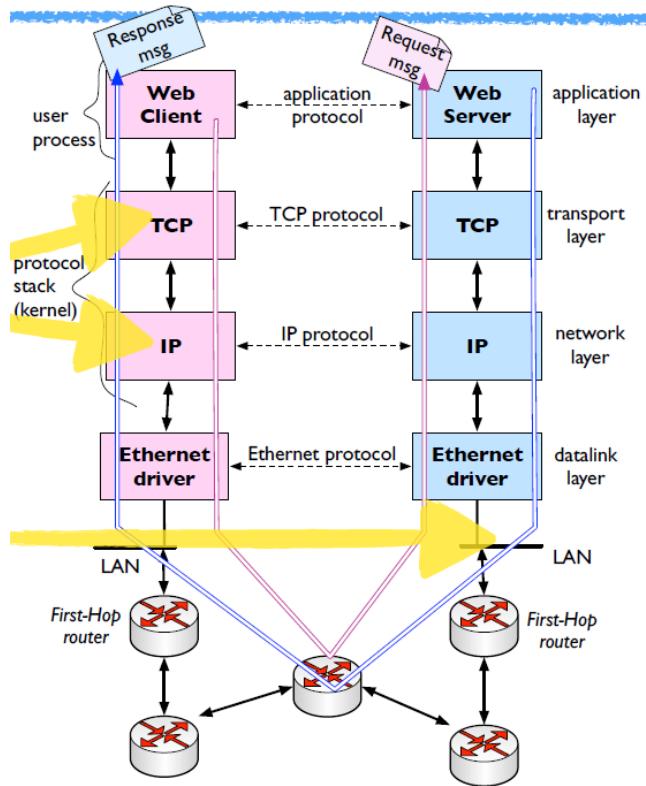
- IP (Internet Protocol): Indirizza i pacchetti di dati e li invia al dispositivo corretto.
  - \* IPv4: La versione più comune (indirizzi a 32 bit).
  - \* IPv6: Versione più recente (indirizzi a 128 bit) per gestire l'esaurimento degli indirizzi IPv4.

- **Livello di Accesso alla Rete:** Si occupa della trasmissione fisica dei dati attraverso i mezzi di comunicazione (cavi, Wi-Fi, ecc.).

Quando un dispositivo invia dati su una rete TCP/IP, avviene quanto segue:

- **Incapsulamento:** I dati vengono suddivisi in pacchetti e arricchiti con informazioni aggiuntive (intestazioni) per ogni livello.
- **Trasmissione:** I pacchetti viaggiano attraverso la rete, passando per router e altri dispositivi, fino alla destinazione.

- **Decapsulamento:** Alla destinazione, ogni livello rimuove la sua intestazione e passa i dati al livello superiore.



## 14.2 Socket creation

In Linux, la funzione socket è usata per creare un socket, che è un punto finale di comunicazione tra due entità (es. client e server) su una rete:

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Restituisce il file descriptor del socket in caso di successo e  $-1$  in caso di errore

dove i parametri sono i seguenti:

- *domain*: Specifica il protocollo di comunicazione e il formato degli indirizzi. I valori comuni sono:

- AF\_INET: Per IPv4
- AF\_INET6: Per IPv6
- AF\_UNIX: Per comunicazione locale tramite file di dominio Unix
- *type*: Determina il tipo di socket:
  - SOCK\_STREAM: Socket orientato alla connessione (es. TCP)
  - SOCK\_DGRAM: Socket senza connessione (es. UDP)
  - SOCK\_RAW: Accesso ai pacchetti IP grezzi
- *protocol*: Specifica il protocollo utilizzato. Di solito si usa 0 per selezionare il protocollo predefinito per il tipo di socket scelto

A questo punto è stata allocata una struttura nel kernel ma senza specificare alcun indirizzo.

### 14.3 Socket Addresses

Un socket address (o indirizzo del socket) è un insieme di informazioni che identifica univocamente un socket in una rete e permette la comunicazione tra dispositivi. Un socket address combina l'indirizzo del dispositivo (IP) e un numero di porta per specificare il servizio sul dispositivo e quindi il punto finale della comunicazione.

In Linux e altre piattaforme Unix, l'indirizzo di un socket è rappresentato usando strutture dati specifiche, come `sockaddr` e le sue varianti:

- **struct sockaddr:**

```
struct sockaddr {
    sa_family_t sa_family; // Famiglia dell'indirizzo (es., AF_INET, AF_INET6)
    char        sa_data[14]; // Dati dell'indirizzo (dipende dalla famiglia)
};
```

- Varianti comuni:

– **struct sockaddr\_in** (per IPv4)

```
struct sockaddr_in {  
    sa_family_t    sin_family;    // Famiglia dell'indirizzo (AF_INET)  
    in_port_t      sin_port;     // Porta (network byte order)  
    struct in_addr sin_addr;    // Indirizzo IPv4  
    char           sin_zero[8]; // Riempimento per allineamento  
};  
  
struct in_addr {  
    uint32_t s_addr;           // Indirizzo IPv4 (network byte order)  
};
```

– **struct sockaddr\_in6** (per IPv6)

```
struct sockaddr_in6 {  
    sa_family_t    sin6_family;   // Famiglia dell'indirizzo (AF_INET6)  
    in_port_t      sin6_port;    // Porta (network byte order)  
    uint32_t       sin6_flowinfo; // Informazioni sul flusso  
    struct in6_addr sin6_addr;  // Indirizzo IPv6  
    uint32_t       sin6_scope_id; // ID di ambito (per indirizzi locali)  
};  
  
struct in6_addr {  
    unsigned char s6_addr[16];   // Indirizzo IPv6 (16 byte)  
};
```

La **funzione inet\_ntop** è utilizzata per convertire un indirizzo IP in forma binaria (network byte order) in una rappresentazione leggibile dall'utente (notazione testuale):

```
#include <arpa/inet.h>
```

```
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

dove i parametri sono:

- *af*: Specifica la famiglia degli indirizzi (AF\_INET per IPv4 e AF\_INET6 per IPv6)
- *src*: Puntatore all'indirizzo IP in formato binario (network byte order) (in\_addr per IPv4 o in6\_addr per IPv6)
- *dst*: Puntatore a un buffer dove sarà memorizzata la rappresentazione testuale dell'indirizzo
- *size*: Dimensione del buffer dst, che deve essere abbastanza grande per contenere la rappresentazione testuale:
  - Per IPv4: almeno INET\_ADDRSTRLEN (16 byte).
  - Per IPv6: almeno INET6\_ADDRSTRLEN (46 byte)

Per effettuare la conversione inversa si utilizza la funzione **inet\_pton**:

```
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

Restituisce 1 in caso di successo, 0 se l'input string non è una valida rappresentazione di un indirizzo IP e -1 in caso di errore

### 14.3.1 Esempio SocketAdress.c

Il seguente codice è un esempio d'uso delle funzione per la manipolazione degli indirizzi IPv4/IPv6 ad opera delle funzioni `inet_ntop()` e `inet_pton()`, oltre alla gestione dei numeri di porta con le funzioni:

- `ntohs()`:
- `htons()`: Converte il numero di porta da host byte order a network byte order ed è sempre necessario farlo prima dello scambio di dati tra hosts

Vediamo di seguito le funzioni una alla volta ed infine il main.

## Inizializzazione di una struttura sockaddr\_in (IPv4)

Il seguente codice è una funzione per **inizializzare una struttura sockaddr\_in** in un'applicazione di rete basata su IPv4. La funzione prende un indirizzo IP in formato stringa e una porta, configurandoli all'interno della struttura sockaddr\_in:

```
int initSockaddr_in(char *ipv4, in_port_t port, struct sockaddr_in *sa){  
    int result = 0; /* valore di ritorno delle Sockets API e della funzione */  
  
    /*  
     * inizializziamo le strutture di indirizzo  
     * - poiché ogni implementazione le definisce arbitrariamente -  
     */  
    memset(sa, 0, sizeof(struct sockaddr_in));  
  
    //avaloriamo il campo Address Family delle strutture di indirizzo  
    sa->sin_family = AF_INET;  
  
    /*  
     * aggiungo alla struct sockaddr_in l'informazione sul numero di porta  
     * ATTENZIONE: anche questo dato deve essere memorizzato in Network Byte Order  
     */  
    sa->sin_port = htons(port);  
  
    /*  
     * Converte l'indirizzo IPv4  
     * espresso in forma dotted-decimal  
     * in un intero a 32 bit (in NETWORK BYTE ORDER),  
     * e lo salva in una struttura in_addr  
     */  
    result = inet_pton(AF_INET, ipv4, &(sa->sin_addr));  
    if (result == 1)  
    {  
        printf("L'indirizzo inserito '%s' è una stringa dotted decimal valida per l'Address Family = %d\n", ipv4, AF_INET);  
    }  
    else if (result == 0)  
    {  
        //address not parseable in the specified address family  
        printf("La stringa '%s' non rappresenta un indirizzo IPv4 valido\n", ipv4);  
    }  
    else  
    {  
        //result == -1  
        perror("inet_pton() error: ");  
    }  
  
    return result;  
}
```

## Stampa di sockaddr\_in (IPv4)

La funzione printSockaddr\_in è progettata per stampare l'indirizzo IP e la porta associata a una struttura sockaddr\_in:

```
void printSockaddr_in(struct sockaddr_in *sa){  
    /*  
     * stringhe in cui verranno memorizzati gli indirizzi IP convertiti in stringa  
     * (/usr/include/netinet/in.h)  
     * INET_ADDRSTRLEN           16  
     * INET6_ADDRSTRLEN          46  
     */  
    char ipv4[INET_ADDRSTRLEN] = {0};  
    const char *ptr = NULL;  
  
    ptr = inet_ntop(AF_INET, &(sa->sin_addr), ipv4, INET_ADDRSTRLEN);  
  
    if (ptr != NULL)  
    {  
        printf("\nIPv4 address = '%s'\n", ipv4);  
    }  
    else  
    {  
        perror("inet_ntop() error: ");  
    }  
  
    printf("Port number = %d\n\n", ntohs(sa->sin_port));  
}
```

## Inizializzazione di una struttura sockaddr\_in6 (IPv6)

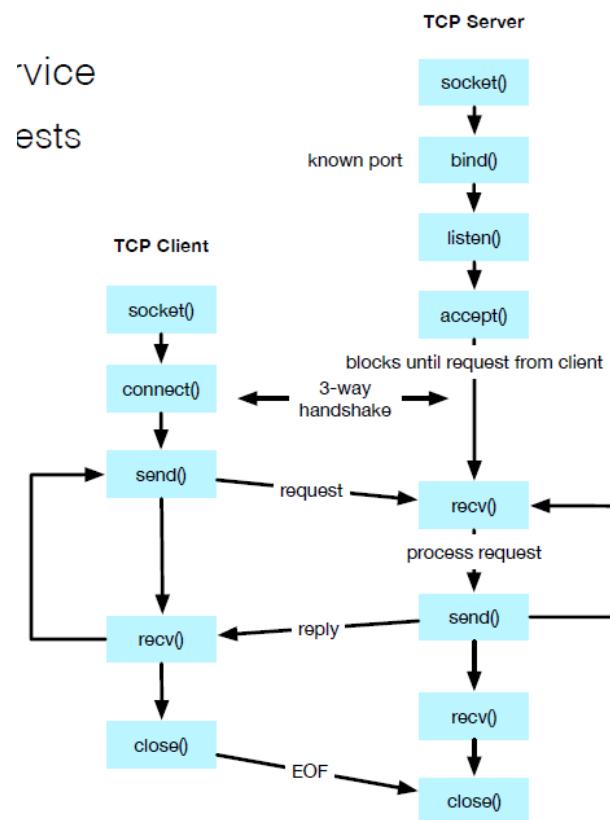
```
int initSockaddr_in6(char *ipv6, in_port_t port, struct sockaddr_in6 *sa){  
    int result = 0; /* valore di ritorno delle Sockets API e della funzione */  
  
    /*  
     * inizializziamo le strutture di indirizzo  
     * - poiché ogni implementazione le definisce arbitrariamente -  
     */  
    memset(sa, 0, sizeof(struct sockaddr_in6));  
  
    //avvaloriamo il campo Address Family delle strutture di indirizzo  
    sa->sin6_family = AF_INET6;  
  
    /*  
     * aggiungo alla struct sockaddr_in6 l'informazione sul numero di porta in Network Byte Order,  
     * da host-byte-order in network byte order  
     */  
    sa->sin6_port = htons(port);  
  
    /*  
     * Converte l'indirizzo IPv6 espresso in hex in valore numerico in NETWORK BYTE ORDER  
     * e lo salva in una struct in6_addr  
     */  
    result = inet_pton(AF_INET6, ipv6, &(sa->sin6_addr));  
    if (result == 1)  
    {  
        printf("L'indirizzo inserito '%s' è una stringa esadecimale valida per l'Address Family = %d\n", ipv6, AF_INET6);  
    }  
    else if (result == 0)  
    {  
        printf("La stringa '%s' non rappresenta un indirizzo IPv6 valido\n", ipv6);  
    }  
    else  
    {  
        //result == -1  
        perror("inet_pton() error: ");  
    }  
  
    return result;  
}
```

## Stampa sockaddr\_in6 (IPv6)

```
void printSockaddr_in6(struct sockaddr_in6 *sa){  
  
    /*  
     * stringhe in cui verranno memorizzati gli indirizzi IP convertiti in stringa  
     * (/usr/include/netinet/in.h)  
     * INET_ADDRSTRLEN          16  
     * INET6_ADDRSTRLEN         46  
     */  
  
    char ipv6[INET6_ADDRSTRLEN] = {0};  
    const char *ptr = NULL;  
  
    ptr = inet_ntop(AF_INET6, &(sa->sin6_addr), ipv6, INET6_ADDRSTRLEN);  
    if (ptr != NULL)  
    {  
        printf("\nIPv6 address = '%s'\n", ipv6);  
    }  
    else  
    {  
        perror("inet_ntop() error: ");  
    }  
  
    //attenzione:  
    printf("la porta in network byte order è %d\n", (int)sa->sin6_port);  
    printf("Port number = %d\n\n", ntohs(sa->sin6_port));  
}
```

## 14.4 TCP Client e TCP Server

Un client TCP e un server TCP sono due entità che comunicano utilizzando il protocollo TCP (Transmission Control Protocol). TCP è un protocollo orientato alla connessione, affidabile e garantisce la consegna dei dati in ordine. Queste entità si scambiano messaggi attraverso un canale di comunicazione che viene stabilito tra di loro.



### 14.4.1 Funzione bind()

Dopo la creazione di un socket, questo deve essere associato un indirizzo IP e una porta specifici, quindi si utilizza la funzione bind:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove *addr* è un puntatore alla struttura `sockaddr` che contiene l'indirizzo e la porta a cui il socket deve essere associato. Di solito, si usa una struttura più specifica come `sockaddr_in` per

indirizzi IPv4 o `sockaddr_in6` per indirizzi IPv6; mentre `addrlen` è la dimensione della struttura dell'indirizzo, che rappresenta quanto è grande la struttura `sockaddr` (si ottiene con `sizeof()`)

#### 14.4.2 Funzione `listen()`

La funzione `listen()` in un'applicazione di rete viene utilizzata per mettere un server socket in modalità "ascolto", in modo che possa accettare le connessioni in ingresso dai client.

```
int listen(int sockfd, int backlog);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove il parametro `backlog` è un intero che specifica la dimensione della coda di connessione (queue) per le connessioni in attesa. Questo parametro indica quante connessioni in attesa possono essere mantenute prima che il server rifiuti nuove richieste di connessione. Quando il server è occupato, le connessioni in ingresso vengono messe in questa coda.

#### 14.4.3 Funzione `accept()`

La funzione `accept()` in un'applicazione di rete è utilizzata in un server per accettare una connessione in ingresso da un client, creando un nuovo socket per gestire la comunicazione con quel client. Questa funzione viene chiamata dopo che il server ha messo il socket in modalità ascolto tramite la funzione `listen()`.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Restituisce un nuovo file descriptor per il socket connesso. Questo nuovo socket è utilizzato per comunicare con il client specifico oppure -1 in caso di errore

#### 14.4.4 Funzione `connect()`

La funzione `connect()` è utilizzata da un client per stabilire una connessione con un server.

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Restituisce 0 in caso di successo e -1 in caso di errore

#### 14.4.5 Funzioni send() e recv()

Le funzioni `send()` e `recv()` sono utilizzate per inviare e ricevere dati attraverso un socket, rispettivamente.

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Restituisce il numero di byte inviati e `-1` in caso di errore

dove `sockfd` è il file descriptor del socket attraverso cui inviare i dati. Questo socket deve essere già connesso (per esempio, tramite la funzione `connect()` per il client o `accept()` per il server), `buf` un puntatore al buffer che contiene i dati da inviare e `flags` è un intero che può essere utilizzato per specificare alcune opzioni speciali per la trasmissione dei dati:

- 0: senza flags, invia i dati normalmente
- `MSG_DONTWAIT`: invia i dati senza bloccare il processo, se il socket è non bloccante
- `MSG_NOSIGNAL`: Se si tenta una scrittura su un socket non più connesso, il flag set chiede di non generare il segnale `SIGPIPE` (il default) ma di impostare invece `errno` su `EPIPE`

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

Restituisce il numero di byte ricevuti e `-1` in caso di errore

dove `sockfd` è il file descriptor del socket attraverso cui ricevere i dati, `buf` è un puntatore al buffer dove verranno memorizzati i dati ricevuti e `flags` è un intero che può essere utilizzato per specificare alcune opzioni speciali per la ricezione dei dati:

- 0: senza flags, riceve i dati normalmente
- `MSG_WAITALL`: aspetta di ricevere l'intero messaggio, se possibile
- `MSG_DONTWAIT`: riceve i dati senza bloccare il processo, se il socket è non bloccante
- `MSG_PEEK`: il kernel permette all'applicazione di leggere i dati nel socket RECV queue ma non li rimuove

#### 14.4.6 Funzione close()

La funzione `close()` è utilizzata per chiudere un socket:

```
int close(int fd);
```

Restituisce 0 in caso di successo e -1 in caso di errore

dove *fd* è il file descriptor di un socket che è stato creato tramite `socket()`, o che è stato ottenuto tramite `accept()` (per un server).

#### 14.4.7 Local e Romote Adress

La funzione `getsockname()` viene utilizzata per ottenere l'indirizzo locale e la porta associati a un socket. È utile per scoprire quale indirizzo IP e quale porta sono stati assegnati a un socket dopo una connessione (tipicamente per un server, che potrebbe non conoscere quale porta è stata assegnata dal sistema operativo)

```
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

La funzione `getpeername()` viene utilizzata per ottenere l'indirizzo e la porta del peer (client o server) a cui il socket è connesso. Questa funzione è utile principalmente nel contesto delle connessioni orientate alla connessione (come quelle TCP), poiché consente di ottenere informazioni sul lato remoto di una connessione.

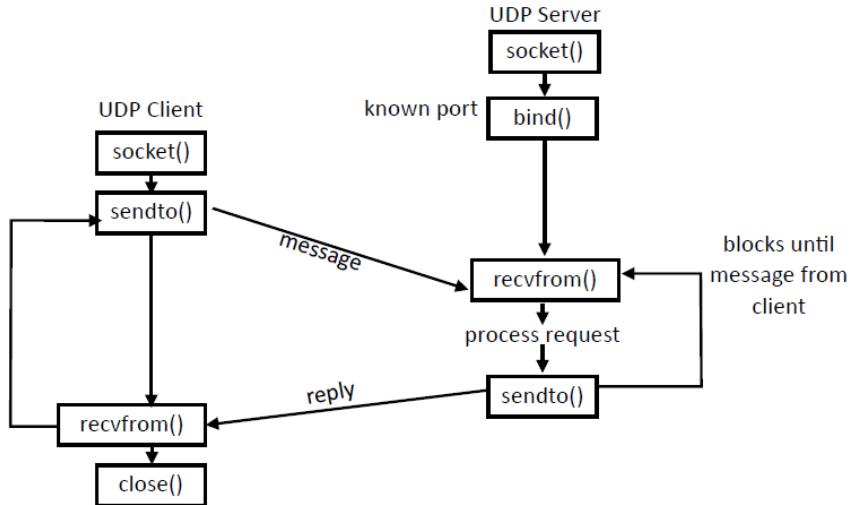
```
int getpeername(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

#### 14.4.8 Esempio TCPServer.c

#### 14.4.9 Esempio TCPClient.c

### 14.5 UDP Client e UDP Server

Un'applicazione UDP (User Datagram Protocol) è senza connessione, il che significa che non richiede una connessione stabile tra il client e il server come accade con TCP. UDP invia pacchetti di dati (detti datagrammi) senza garantire che arrivino a destinazione.



#### 14.5.1 Funzione sendto()

La funzione `sendto()` è utilizzata per inviare un messaggio tramite un socket in modo esplicito verso un determinato indirizzo e porta. È comunemente utilizzata nei programmi UDP, dove non c'è una connessione stabilita e ogni messaggio deve specificare il destinatario. Al contrario dell'UDP, infatti, un messaggio può essere inviato verso diverse destinazioni:

```
ssize_t sendto(int sockfd, const void *buffer, size_t len,
               int flags, const struct sockaddr *dest_addr, socklen_t addrlen);
```

Restituisce il numero di byte inviati, oppure `-1`

dove *buffer* è un puntatore ai dati da inviare, *dest\_addr* è un puntatore ad una struttura `sockaddr` che specifica l'indirizzo del destinatario (può essere `sockaddr_in` o `sockaddr_in6`) e quindi *addrlen* indica la lunghezza della struttura *dest\_addr*.

Di seguito un flag che potrebbe essere utile:

- `MSG_NOSIGNAL`: Se si tenta una scrittura su un socket non più connesso, il flag set chiede di non generare il segnale `SIGPIPE` (il default) ma di impostare invece `errno` su `EPIPE`

### 14.5.2 Funzione recvfrom()

La funzione recvfrom() viene utilizzata per ricevere dati da un socket in modo esplicito, ottenendo anche informazioni sull'indirizzo del mittente:

```
ssize_t recvfrom(int sockfd, void *buffer, size_t len, int flags,  
                  struct sockaddr *src_addr, socklen_t *addrlen);
```

Restituisce il numero di byte ricevuto o -1

dove *sock* indica il file descriptor da cui ricevere i dati, *buffer* è un puntatore a un'area di memoria dove verranno memorizzati i dati ricevuti, *src\_addr* è un puntatore a una struttura *sockaddr* che memorizzerà l'indirizzo del mittente. Se non si è interessati a questa informazione, si può passare NULL.

UDP è un protocollo senza connessione. Se il server non è in esecuzione a funzione recvfrom() rimarrà bloccata (se il socket è bloccante) o fallirà con EAGAIN (se il socket è non bloccante).

Di seguito un flag che potrebbe essere utile:

- **MSG\_PEEK**: il kernel permette all'applicazione di leggere i dati nel socket RECV queue ma non li rimuove

### 14.5.3 Esempio UDPServer.c

### 14.5.4 Esempio UDPClient.c

## 14.6 Esempio showip.c

Si trova nel pdf Beej's Guide to Network a pagina 24 (Messo nella dir SNP2024)

## 14.7 Esempio server.c

Si trova nel pdf Beej's Guide to Network a pagina 32 (Messo nella dir SNP2024)

## 14.8 Hostname Resolution

La risoluzione del nome host è il processo con cui un nome di dominio (ad esempio www.example.com) viene tradotto in un indirizzo IP (ad esempio 192.168.1.1) che può essere utilizzato dai protocolli di rete per instradare i dati attraverso Internet o una rete locale.

Questo processo è gestito dal Sistema dei Nomi di Dominio (DNS), che funziona nel seguente modo:

- **Query DNS:** Quando un client (ad esempio un browser web o un server) cerca di raggiungere un dominio, invia una query DNS a un resolver DNS. La query richiede l'indirizzo IP associato a un determinato nome host.
- **Controllo nella Cache:** Il resolver DNS prima verifica nella propria cache per vedere se ha già l'indirizzo IP per il nome host in questione, relativo a una query precedente. Se trovato, restituisce l'indirizzo IP memorizzato nella cache al client.
- **Query DNS Ricorsiva:** Se il resolver DNS non ha l'indirizzo IP nella cache, esegue una query ricorsiva. Inizia chiedendo ai server root DNS informazioni sui server del dominio di livello superiore (TLD) (ad esempio .com o .org).
- **Riferimento ai Server Autorevoli:** I server root rimandano la query ai server DNS autorevoli per il TLD del dominio (come example.com per il dominio www.example.com). I server autorevoli forniscono quindi l'indirizzo IP.
- **Restituzione del Risultato:** Il resolver DNS memorizza l'indirizzo IP nella cache e lo invia al client. Il client può ora utilizzare questo indirizzo IP per stabilire una connessione con il server.

## 14.9 Funzione getaddrinfo()

La funzione `getaddrinfo()` è una funzione di rete in C che serve per ottenere informazioni su come connettersi a un indirizzo specifico (come un host o un servizio), come l'indirizzo IP di un server o le informazioni sui socket.

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node, const char *service, const struct
                addrinfo *hints, struct addrinfo **res);

```

Restituisce 0 in caso di successo e un codice di errore in caso di insuccesso

Il parametro *node* rappresenta il nome dell'host (come un dominio, ad esempio "www.example.com") o un indirizzo IP (ad esempio "192.168.1.1"). Può essere NULL se si vuole solo ottenere informazioni sul servizio. Il parametro *service* rappresenta il servizio (come "http" o una porta specifica, ad esempio "80"). Può essere NULL se si vuole solo ottenere informazioni sull'host. Il parametro *hints* è un puntatore a una struttura **addrinfo** che specifica preferenze per il tipo di socket che si sta cercando. Questa struttura può essere impostata per ottenere risultati più specifici, come la famiglia di indirizzi (IPv4 o IPv6). Il parametro *res* è un puntatore a un puntatore a una lista collegata di strutture **addrinfo** che conterranno i risultati della funzione. Ogni struttura **addrinfo** contiene informazioni su un indirizzo.

La struttura **addrinfo** contiene informazioni sull'indirizzo, come l'indirizzo IP e il tipo di socket.

```

struct addrinfo {
    int          ai_flags;      // Flagi di ricerca (ad esempio, AI_PASSIVE)
    int          ai_family;     // Famiglia di indirizzi (AF_INET, AF_INET6)
    int          ai_socktype;   // Tipo di socket (SOCK_STREAM, SOCK_DGRAM)
    int          ai_protocol;   // Protocollo di comunicazione
    size_t       ai_addrlen;    // Lunghezza dell'indirizzo
    struct sockaddr *ai_addr;   // Puntatore all'indirizzo
    char         *ai_canonname; // Nome canonico dell'host
    struct addrinfo *ai_next;   // Puntatore alla struttura successiva
};

```

## 14.10 Funzione freeaddrinfo()

La funzione `freeaddrinfo()` in C viene utilizzata per liberare la memoria precedentemente allocata dalla funzione `getaddrinfo()` quando non è più necessaria. Poiché `getaddrinfo()` restituisce una lista collegata di strutture `addrinfo`, è importante utilizzare `freeaddrinfo()` per deallocare correttamente questa memoria e evitare perdite di memoria (memory leaks).

```
void freeaddrinfo(struct addrinfo *res);
```

## 14.11 Funzione getnameinfo()

La funzione `getnameinfo()` è una funzione di rete in C che consente di ottenere il nome di un host o il nome di un servizio a partire da un indirizzo IP (o una struttura di socket) e da un numero di porta. È l'opposto di `getaddrinfo()` e viene utilizzata per la risoluzione inversa, cioè per ottenere il nome di dominio di un host a partire dal suo indirizzo IP o per ottenere il nome di un servizio dato un numero di porta.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host,
                socklen_t hostlen, char *service,
                socklen_t servicelen, int flags);
```

Restituisce 0 in caso di successo oppure un codice di errore in caso di insuccesso

Il parametro *sa* è un puntatore a una struttura `sockaddr` che contiene l'indirizzo di rete (ad esempio, un indirizzo IPv4 o IPv6). Il parametro *salen* è la lunghezza della struttura `sockaddr` a cui punta *sa*. Può essere `sizeof(struct sockaddr_in)` per un indirizzo IPv4, ad esempio. Il parametro *host* è un buffer in cui verrà scritto il nome di dominio dell'host risolto (ad esempio, "www.example.com"). Deve avere una lunghezza sufficiente a contenere il nome completo, ad

esempio NI\_MAXHOST che è una costante definita in <netdb.h>. Il parametro *hostlen* è la lunghezza del buffer host (NI\_MAXHOST). Il parametro *service* è un buffer in cui verrà scritto il nome del servizio (ad esempio, "http" o "ftp") o il numero di porta. Il parametro *servicelen* è la lunghezza del buffer service (NI\_MAXSERV). Infine può essere specificato un insieme di flag:

- NI\_NUMERICHOST: Se specificato, la funzione restituirà l'indirizzo numerico dell'host, invece di cercare di risolverlo in un nome di dominio
- NI\_NUMERICSERV: Se specificato, la funzione restituirà il numero di porta invece del nome del servizio.
- NI\_DGRAM: Se specificato, la funzione tratterà l'indirizzo come se fosse un datagram (UDP), anziché un flusso (TCP).

## 14.12 Funzione gai\_strerror()

La funzione gai\_strerror() in C viene utilizzata per ottenere una stringa di errore descrittiva a partire da un codice di errore restituito dalle funzioni di risoluzione dei nomi come getaddrinfo() o getnameinfo(). Questa funzione fornisce una spiegazione leggibile dell'errore che si è verificato, facilitando la gestione degli errori nella programmazione di rete.

```
#include <netdb.h>

const char *gai_strerror(int ecode);
```

Di seguito un esempio del suo utilizzo:

```
// Chiamiamo getaddrinfo per ottenere l'indirizzo

status = getaddrinfo("www.nonexistentwebsite.com", "http", &hints, &res);

if (status != 0) {

    // In caso di errore, stampiamo la descrizione dell'errore
    fprintf(stderr, "Errore: %s\n", gai_strerror(status));

    return 1;
}
```