Programmazione concorrente nei sistemi Unix

Giuseppe Lipari, Luca Abeni, Antonino Casile
Scuola Superiore di Studi Universitari e di Perfezionamento
S. Anna

11 novembre 2017

Indice

1. Introduzione **5**
- 1.1. Sistemi Multiprogrammati . . . 5
  - 1.1.1. Concetti Fondamentali . . . 5
- 1.2. Supporto di Sistema . . . 6
- 1.3. Note generali su Unix . . . 6

2. Processi **8**
- 2.1. Struttura di un processo . . . 8
- 2.2. Struttura di un programma C . . . 8
  - 2.2.1. Variabili di ambiente . . . 10
- 2.3. Layout di memoria di un processo . . . 11

3. Il File System di Unix **13**
- 3.1. Concetti fondamentali . . . 13
- 3.2. Files . . . 14
- 3.3. La libreria Posix per i file . . . 15
- 3.4. Strutture dati coinvolte . . . 17
  - 3.4.1. Funzione dup() . . . 19
- 3.5. Permessi e proprietà di un file . . . 19
  - 3.5.1. File permission bits . . . 20
- 3.6. La I/O Standard Library . . . 22
  - 3.6.1. Relazioni tra libreria Posix e libreria ANSI . . . 24

4. Multiprocessing **26**
- 4.1. Processi . . . 26
  - 4.1.1. Identificatori di processo . . . 26
  - 4.1.2. Creazione di nuovi processi . . . 27
  - 4.1.3. Terminazione dei processi . . . 29
  - 4.1.4. Caricamento di un programma in memoria . . . 31
  - 4.1.5. Un esempio . . . 33
- 4.2. Esercizi . . . 35

5. Sincronizzazione fra processi **36**
- 5.1. I segnali . . . 36
- 5.2. Specifica di un signal handler . . . 37

5.3. Invio di un segnale . . . 41
5.4. Sospensione in attesa di un segnale . . . 43
5.5. Primitiva alarm(), e settaggio dei timeout . . . 47
5.6. Il segnale SIGCHLD . . . 48
5.7. Comunicazione fra processi . . . 50
5.7.1. Comunicazione tramite pipe . . . 50
5.7.2. Un Esempio . . . 52
5.7.3. Comunicazione tramite FIFO . . . 55
5.7.4. Le IPC di System V . . . 57
5.7.5. Un esempio . . . 59
5.8. Esercizi . . . 63
5.8.1. Segnali . . . 63
5.8.2. Pipe . . . 63
5.8.3. Pipe con nome . . . 64
5.8.4. Vari . . . 64

6. Multithreading **65**
6.1. POSIX threads . . . 65
6.1.1. Compilazione di un programma multithreaded . . . 66
6.1.2. Creazione dei Thread . . . 66
6.1.3. Esempio di creazione di due thread . . . 69
6.1.4. Attributi di creazione di un thread . . . 70
6.1.5. Passaggio parametri . . . 72
6.2. Meccanismi di mutua esclusione . . . 72
6.2.1. Esempio: meccanismi di mutua esclusione . . . 74
6.3. Attesa di una condizione . . . 77
6.3.1. Un esempio: meccanismi di sincronizzazione su una condizione . . . 78
6.4. Semafori . . . 81
6.4.1. Un esempio . . . 82
6.5. Punti di cancellazione . . . 84

7. Sistemi distribuiti **87**
7.1. Modello client/server . . . 87
7.2. L'interfaccia dei socket . . . 87
7.2.1. Socket TCP/IP . . . 88
7.2.2. Esempio . . . 94
7.3. Server concorrenti . . . 98
7.3.1. Server multiprocesso . . . 99
7.3.2. Server multithread . . . 100

A. Funzioni di utilità **104**

Elenco delle figure

2.1. Sequenza di chiamate fatte all'entrata e all'uscita di un processo. 10
2.2. Tipico layout di memoria di un processo. 11

3.1. Esempio di file system. 15
3.2. Strutture dati nel kernel. 18
3.3. La struttura `struct stat`. 20
3.4. Esempio di output del comando `ls -l`. 21

7.1. Passi necessari a stabilire una connessione TCP/IP 89

1. Introduzione

1.1. Sistemi Multiprogrammati

Un sistema multiprogrammato è un sistema che permette l'esecuzione simultanea di più programmi, che a loro volta possono essere composti da vari *processi* o *thread*. Questo può permettere a più utenti di accedere al sistema contemporaneamente, o ad uno stesso utente di eseguire più programmi simultaneamente (aumentando l'utilizzabilità del sistema), o ad un singolo programma di scomporre la propria attività in un insieme di attività concorrenti (semplificando la struttura logica del programma).

Si passa così dal concetto di programma sequenziale al concetto di programma parallelo, utilizzando un paradigma di programmazione concorrente. Questo, se da un lato permette di strutturare meglio i programmi, dall'altro richiede una particolare attenzione a problemi di sincronizzazione fra le varie attività (spesso l'esecuzione di un programma concorrente risulta non deterministica) e una stretta interazione con il sistema operativo.

Prima di passare a vedere quali sono le funzionalità offerte dal sistema operativo a supporto della programmazione concorrente e come utilizzarle, è bene andare a definire in maniera più formale alcuni concetti di base.

1.1.1. Concetti Fondamentali

Come accennato, un sistema concorrente permette l'esecuzione contemporanea di varie attività tramite multiprocessing o multithreading: andiamo allora a capire cosa si intende con questi concetti.

Si definisce *algoritmo* il procedimento logico seguito per risolvere un determinato problema. Tale algoritmo può essere descritto tramite un opportuno formalismo (chiamato *linguaggio di programmazione*), in modo da poter essere eseguito su un elaboratore. Tale codifica di un algoritmo, espressa tramite un linguaggio di programmazione è detta *programma*.

L'esecuzione di un programma può essere di tipo sequenziale, oppure può essere composta da più attività che eseguono in parallelo. In questo secondo caso (programma concorrente), le varie attività che eseguono in parallelo devono essere opportunamente sincronizzate fra loro e necessitano di comunicare e cooperare al fine di portare a termine il proprio scopo. Il modo in cui tali attività concorrenti cooperano e comunicano le distingue fra *processi* e *thread*.

I processi sono caratterizzati da un insieme di risorse private, fra cui gli spazi di memoria, per cui un processo non può accedere allo spazio di memoria di un altro per

comunicare con esso. Per questo motivo, i processi si sincronizzano e comunicano fra loro tramite *scambio di messaggi*. Viceversa i thread condividono varie risorse, fra cui lo spazio di memoria, per cui possono comunicare tramite *memoria comune*, e sincronizzarsi tramite semafori.

1.2. Supporto di Sistema

Come noto, il sistema operativo è il programma che si occupa di gestire le risorse hardware della macchina, ivi comprese anche la CPU e la memoria (fisica e virtuale). Quindi, il sistema operativo deve fornire anche un supporto (più o meno esteso) alla concorrenza: per poter creare processi o thread, per sincronizzarli e per farli comunicare, il programma deve richiedere tali funzioni al sistema operativo, tramite specifiche chiamate di sistema, o *system call* (brevemente, *syscall*).

Spesso, l'insieme delle syscall esportate dal sistema operativo, detto anche interfaccia, o più propriamente *Application Programming Interface* (API), risulta essere troppo di basso livello per essere utilizzabile in maniera semplice dall'utente. Per questo motivo, vengono talvolta fornite delle librerie che implementano funzionalità di più alto livello basandosi sulle chiamate di sistema. Tipicamente, una libreria mette a disposizione delle chiamate di libreria (*library call*, o più brevemente *libcall*) che esportano un'interfaccia più potente e semplice da usare rispetto a quella fornita dalle chiamate di sistema. Bisogna però tenere presente che una chiamata di libreria eseguirà del codice di libreria in modo utente, e poi eventualmente chiamerà una o più syscall che eseguiranno in modo kernel.

Un tipico esempio per chiarire quanto appena detto è costituito dalla libreria standard del linguaggio C (`libc`): fra le altre cose, essa implementa gli stream di I/O, fornendo un modo semplice e potente per accedere ai file. Un file può essere aperto tramite la libcall `fopen()` e può essere acceduto in scrittura tramite la libcall `fprintf()`, che permette di scrivere numeri e stringhe effettuando automaticamente la conversione di formato. Tali funzioni si basano su alcune chiamate di sistema, che in sistemi Unix sono `open()` e `write()`, le quali permettono un accesso al file senza bufferizzazione dello stream e senza conversione di formato. Un utente può indifferentemente scegliere di accedere ad un file tramite le chiamate di sistema o le chiamate della `libc`, ma deve sempre sapere cosa sta facendo (e cosa la propria scelta implica), in quanto i due tipi di accesso non possono essere arbitrariamente mescolati (per esempio, non si può - chiaramente - effettuare una `fprintf()` su un file aperto con `open()`).

1.3. Note generali su Unix

Nel seguito analizzeremo le syscall disponibili a supporto della multiprogrammazione, ed in particolare vedremo come creare differenti flussi di esecuzione all'interno di un programma, come sincronizzarli e come realizzare la comunicazione fra thread e processi. Tali syscall dipendono dal sistema operativo, ed in questa sede faremo riferimento ai sistemi operativi Unix-like (Linux in particolare), secondo lo standard POSIX.

Sebbene il supporto alla multiprogrammazione fornito dal sistema sia sfruttabile utilizzando diversi linguaggi di programmazione, in questa sede utilizzeremo il linguaggio C. Si assume che il lettore sia familiare con i costrutti di base di tale linguaggio e con le librerie standard da esso fornite. In ogni caso, nel capitolo 2 verrà fatta una brevissima introduzione sulla struttura di un programma C, mentre nel capitolo 3 verrà presentata brevemente la I/O Standard Library.

Ogni system call ritorna un intero, che in caso di valore negativo segnala una condizione di errore: è allora buona norma verificare che tale valore sia maggiore o uguale a 0 ogni volta che si invoca una syscall, utilizzando per esempio un pezzo di codice simile al seguente.

```
int res;

res = syscall(...);
if (res < 0) {
  perror("Error calling syscall");
  exit(-1);
}
// Program continues here...
```

La funzione `perror()` invia sullo standard error (tipicamente, il video) un messaggio di errore composto dalla stringa passata come parametro seguita da una descrizione del motivo per cui l'ultima syscall chiamata è fallita.

I prototipi delle system call e le definizioni di costanti e strutture dati ad esse necessarie sono contenuti in alcuni header file di sistema (generalmente nella directory `/usr/include/sys`); per poter utilizzare una determinata syscall bisogna quindi includere gli adeguati header. I sistemi Unix mettono a disposizione il comando `man`, che fornisce una breve descrizione della semantica di una syscall specificata, assieme a specificarne la sintassi ed a elencare gli header da includere per utilizzare tale syscall. **Riferirsi sempre alle manpage per includere i file corretti ed utilizzare una syscall con la giusta sintassi.**

2. Processi

2.1. Struttura di un processo

Un **programma** è un file eseguibile residente sul disco. Può essere eseguito tramite un comando di shell oppure invocando una funzione *exec()*. Un **processo** è una istanza del progamma che sta eseguendo[1]. Ogni processo ha un identificatore unico nel sistema, detto **process ID**, che è un intero non negativo.

Quindi, se lanciamo due volte lo stesso programma otteniamo 2 processi distinti, ognuno con il suo process ID. Un processo può ottenere il suo ID invocando la primitiva *getpid()*. Nel capitolo 4 presenteremo meglio il concetto di processo nel contesto della multi-programmazione. In questo capitolo ci concentreremo sul layout di memoria di un processo, e sulle sue interazioni con il sistema operativo.

Dato che il linguaggio più usato in assoluto nei sistemi Unix è il C, e dato che tutti i prototipi delle chiamate di sistema sono espresse in C, vale la pena fare qualche richiamo sulla struttura di un programma scritto in C.

2.2. Struttura di un programma C

Ogni programma scritto in C ha un entry-point che è la funzione main. Il suo prototipo più generale è:

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

Il primo parametro rappresenta il numero di argomenti passati al programma sulla linea di comando, mentre il secondo parametro è il puntatore a un array di stringhe contenente gli argomenti veri e propri. Il primo argomento (contenuto nella stringa argv[0]) è sempre il nome del programma. Quindi argc vale sempre almeno 1.

Un processo può terminare in 5 modi diversi, 3 sono normali e 2 sono anormali:

- Terminazione normale:
  - eseguendo l'istuzione return dal main;
  - chiamando exit();
  - chiamando _exit().

[1] A volte un processo viene chiamato *task*: dato che task è un termine più generico, nel seguito cercheremo sempre di usare il termine processo

- Terminazione anormale:
  - chiamando direttamente abort();
  - ricevendo un segnale non gestito.

I prototipi delle due funzioni exit sono elencati di seguito:

```
#include <stdlib.h>

void exit(int status);
```

```
#include <unistd.h>

void _exit(int status);
```

La exit() fa una "pulizia" dello stato del processo prima di uscire (tipicamente, chiude tutti i file e i descrittori aperti), mentre _exit() torna brutalmente al sistema. È anche possibile installare delle funzioni da chiamare prima dell'uscita tramite la funzione atexit().

```
#include <stdlib.h>

void atexit(void (*func)(void));
```

Ecco un esempio di utilizzo:

```
#include <stdlib.h>

void myexit(void);

int main()
{
  printf("Hello world\n");
  atexit(myexit);
  printf("Just before exiting ...\n");
  return 0;
}
```

La sequenza di chiamate fatte quando viene lanciato un processo, e mentre esso è in esecuzione, è riassunta in Figura 2.1: quando viene lanciato un processo, per prima cosa parte una routine di inizializzazione (non visibile all'utente), chiamata *C Startup Routine*. Poi, viene invocata la funzione main(). Se la funzione esce con return, si ritorna alla *C Startup Routine*, la quale invoca exit() passandogli il valore di ritorno ottenuto dal main. Il main (o una delle funzioni invocate dal main) a sua volta può invocare direttamente la funzione exit(), oppure la _exit(). La differenza è che la exit() fa una serie di cose, tra le quali invocare gli handler, ovvero quelle funzioni che sono state installate con la atexit(). Infine, sia la _exit() che la exit() ritornano al kernel, il quale fa un'ulteriore lavoro di *pulizia*, cancellando quasi tutte le strutture interne relative al processo. Nel capitolo 4, vedremo che in realtà alcune strutture rimangono comunque presenti fino a che il processo padre non chiama la wait().

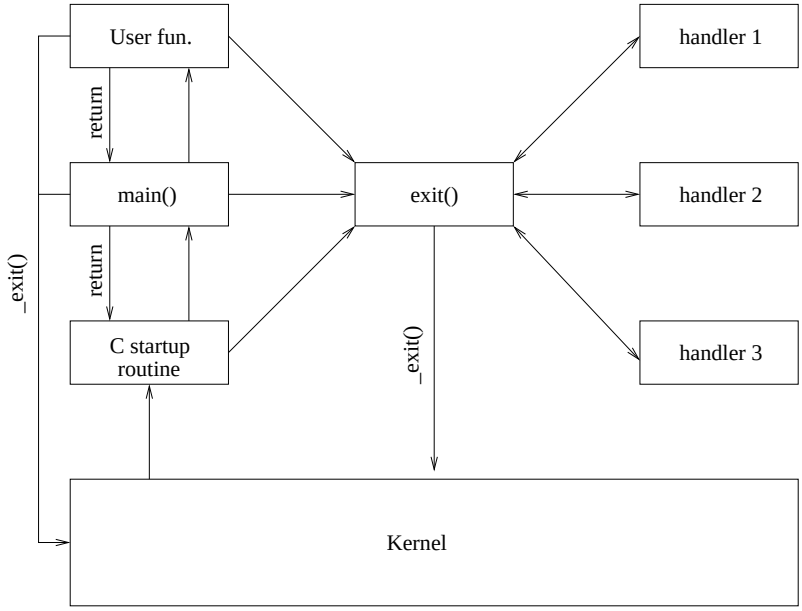

Figura 2.1.: Sequenza di chiamate fatte all'entrata e all'uscita di un processo.

2.2.1. Variabili di ambiente

Ogni shell prevede un modo per settare delle variabili di ambiente. Su bash:

```
export VARNAME=valore
```

Ad esempio, una tipica variabile di ambiente è la variabile `PATH`: si tratta di una sequenza di directory separate dal carattere `:`, e ogni volta che si digita un comando, la shell cerca il file eseguibile nelle directory elencate nel `PATH`. Sempre con la bash, per vedere tutto le variabili definite e il loro contenuto, basta digitare il comando `export`.

È possibile leggere da programma le variabili d'ambiente, che sono memorizzate come un array di stringhe, tramite la funzione getenv():

```
#include <stdlib.h>

char *getenv(const char *name);
```

È infine possibile modificare una variabile di ambiente tramite le seguenti chiamate di funzione, dal significato abbastanza intuitivo:

```
int setenv(const char *name, const char *value, int rewrite);
void unsetenv(const char *name);
int putenv(const char *str);
```

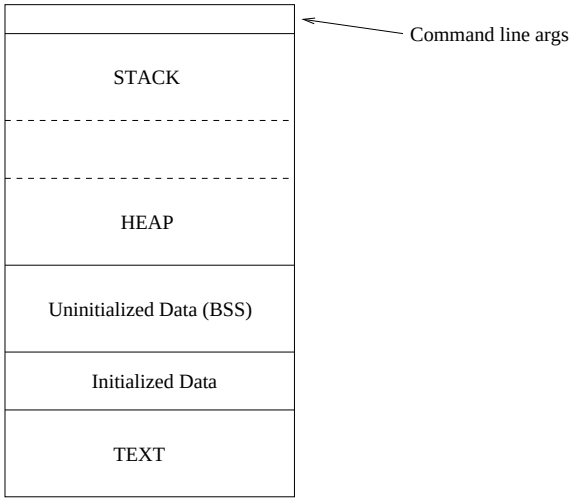

Figura 2.2.: Tipico layout di memoria di un processo.

2.3. Layout di memoria di un processo

Ogni processo ha un suo spazio di indirizzamento privato e non visibile dall'esterno. Ciò vuol dire che due processi non possono accedere a una stessa zona di memoria[2]. Questa separazione totale degli spazi di indirizzamento è ottenuta sfruttando le caratteristiche hardware del processore (segmentazione, protezione di memoria, ecc.).

Lo spazio di indirizzamento di un processo è di solito logicamente organizzato come in figura 2.2. Naturalmente, l'esatto layout dipende dal processore utilizzato e dalle convenzioni del sistema operativo e del compilatore C. Per esempio, in certi sistemi lo stack cresce vero l'alto, al contrario di come mostrato in figura.

Si distingono:

- una zona di memoria (o *segmento*) TEXT, che contiene il codice del programma;
- un segmento di dati inizializzati;
- un segmento di dati non inizializzati (spesso indicato con BSS); di solito il loader si incarica di inizializzarlo a zero;
- un segmento a comune fra lo stack e lo heap;

In particolare, lo heap è quella zona di memoria a lunghezza variabile in cui vengono allocate le zone di memoria che l'utente alloca dinamicamente con le funzioni malloc e free:

[2] Ciò è in realtà possibile possibile tramite la primitiva mmap, che non viene presentata in questo corso per motivi di tempo.

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void * ptr);
```

Queste funzioni non sono chiamate di sistema operativo (syscall), ma sono implementate nella C standard library, e dunque nello spazio utente (libcall). Esse comunque si appoggiano su delle chiamate di sistema (tipicamente la sbrk()), che permetteno di allargare o restringere la dimensione dello heap.

Bisogna fare molta attenzione nell'uso di queste due funzioni! Un problema abbastanza grosso è che le due funzioni in questione *non sono rientranti*, quindi non dovrebbero essere usate all'interno di signal handler, e comunque bisognerebbe utilizzarle con attenzione in presenza di segnali.

3. Il File System di Unix

3.1. Concetti fondamentali

I sitemi Unix sono sistemi multi-utenti. Ogni utente abilitato nel sistema ha a disposizione un certo numero di risorse (un certo spazio disco, la possibilità di eseguire certi programmi ecc.), dette genericamente **account**. Ogni utente nel sistema è identificato da:

User ID : è un intero unico e non negativo. L'utente con ID pari a 0 è il *superuser*, ovvero l'amministratore di sistema.

Login Name : è una stringa anch'essa unica nel sistema. Il login name del super user è root, quindi nel seguito utilizzeremo indifferentemente root o superuser per indicare l'amministratore di sistema.

Password : ogni utente ha una password che gli viene richiesta al momento del login nel sistema. Non è necessariamente unica, nel senso che due utenti potrebbero avere la stessa password.

Group ID : identifica il gruppo a cui l'utente appartiene. Appartenere a un certo gruppo piuttosto che a un altro abilita certi permessi, come vedremo nel seguito.

Home Directory : è la directory dove vengono conservati i file privati dell'utente.

Shell Iniziale : è la *shell* (o interprete di comandi) che viene lanciata dopo la procedura di login.

Un utente può accedere al sistema tramite la *procedura di login*. Ad esempio, quando l'utente accede a un terminale collegato al sistema, si presenta una schermata in cui viene richiesto di inserire il *login name* e successivamente la *password*. Se il nome e la password inserite corrispondono a uno degli account registrati nel sistema, allora viene eseguito il processo *shell* che si posiziona sulla directory iniziale. A questo punto, l'utente si trova con un *prompt* che segnala la disponibilità del programma *shell* ad accettare comandi, come ad esempio ottenere la lista dei file della directory corrente con il programma `ls`, lanciare un programma utente, e così via.

Le informazioni elencate precedentemente (esclusa la password) si trovano in un file chiamato /etc/passwd, che può essere acceduto in lettura da tutti e in scrittura solo dal root. Un esempio di file passwd è mostrato di seguito: ogni linea contiene le informazioni

per ogni utente separate dal carattere ':'. Gli unici utenti "veri" nell'esempio sono root, lipari e elena.

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:
operator:x:11:0:operator:/root:
games:x:12:100:games:/usr/games:
gopher:x:13:30:gopher:/usr/lib/gopher-data:
ftp:x:14:50:FTP User:/home/ftp:
nobody:x:99:99:Nobody:/:
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
lipari:x:500:500::/home/lipari:/bin/bash
elena:x:0:0:elena:/home/elena:/bin/bash
```

Nei vecchi sistemi le password degli utenti venivano anch'esse memorizzate nel file /etc/passwd *criptate tramite un algoritmo unidirezionale. In questo modo, quando l'utente inseriva la password, la procedura di login applicava l'algoritmo ricavando una stringa che confrontava con il contenuto il campo opportuno del file* /etc/passwd*, mentre era impossibile dalla stringa criptata risalire alla password originale.*

Però, un eventuale hacker, nell'ottica di scoprire la password di alcuni utenti, poteva comunque provare ad applicare l'algoritmo per criptare (che è pubblico) su una serie di password possibili, e confrontare i risultati con il contenuto del file. Con la potenza dei moderni computer era solo questione di tempo prima di riuscire a ricavare la password del root*.*

Quindi, per maggior sicurezza, nei sistemi moderni le password e alcune altre informazioni sono contenute nel file /etc/shadow *(in alcuni sistemi chiamato* master.passwd*) che può essere letto o scritto solo da* root*, e quindi l'attacco descritto sopra non è possibile.*

3.2. Files

Un nome di file è una stringa che può contenere ogni carattere esclusi '/' e '\0'. Tradizionalmente il file system è organizzato ad albero: c'è una directory principale (indicata con "/") ; ogni directory può contenere file o altre directory, in maniera ricorsiva. Ogni file quindi è contenuto in una sola directory, detta directory padre del file. In figura 3.1 c'è un esempio di struttura di directory presa da una famosa distribuzione di Linux.

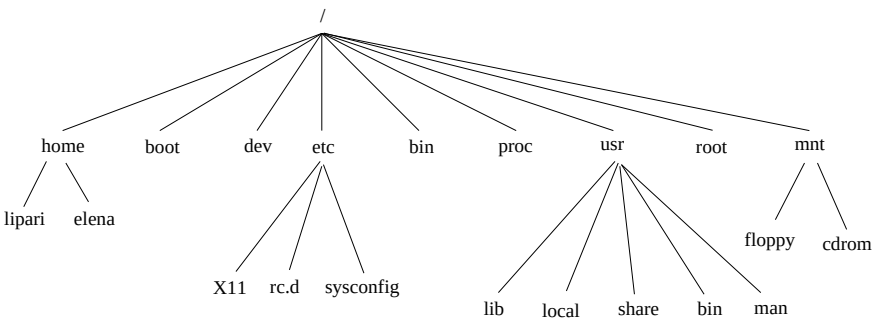

Figura 3.1.: Esempio di file system.

In Unix una directory è un file speciale, che contiene una lista dei file contenuti nella directory[1]. La directory è leggibile da tutti quelli che hanno il permesso in lettura sulla directory, ma è scrivibile solo dal kernel tramite le apposite primitive, in modo da garantire la consistenza del file system.

Il **full pathname** di un file è la sequenza delle directory, a partire da quella principale, per arrivare al file. Quindi un full pathname comincia sempre per '/'.

La **current working directory** (cwd) è la directory corrente di un processo. Dato che la shell è un processo, essa possiede una cwd che è possibile cambiare con il comando di shell cd. Quando un utente entra nel sistema, la procedura di login lancia una shell la cui cwd è la home directory dell'utente. Per ottenere il pathname della cwd esiste il comando pwd.

Il **relative pathname** di un file è la sequenza delle directory, a partire da quella corrente, per arrivare al file. In un pathname (full o relative), il simbolo . indica la directory corrente e il simbolo .. la directory padre.

Esempio 1. Se la cwd è `/home/lipari/bin`, il pathname `../rtlib/src/task.c` indica il file `task.c` nella directory `/home/lipari/rtlib/src/`, mentre il pathname `./killns.pl` indica il file `killns.pl` nella directory `/home/lipari/bin`.

3.3. La libreria Posix per i file

In questa sezione e nelle seguenti, descriveremo la libreria standard POSIX per operare sui file.

Un **descrittore di file** è un intero non negativo che individua un file aperto da un processo. Ogni processo possiede una tabella dei descrittori che associa ogni descrittore aperto per il processo a un file. Di solito, il descrittore 0 corrisponde al file di

[1] Cosa contenga questo file speciale è dipendente dall'implementazione e non c'è alcuno standard al riguardo.

Macro	Modalità
O_RDONLY	solo in lettura
O_WRONLY	solo in scrittura
O_RDWR	in lettura/scrittura
O_APPEND	in append
O_CREAT	crea il file se non esiste
O_EXCL	chiudi in caso di exec
O_TRUNC	tronca la lunghezza del file a 0
O_NONBLOCK	le operazioni sono non bloccanti
O_SYNC	la write non ritorna finché il dato non è effettivamente scritto sul disco.

Tabella 3.1.: flags per la open().

standard input, il descrittore 1 al file di *standard output* e il descrittore 2 al file di *standard error*. Per esigenze di standardizzazione, POSIX definisce le macro `STDIN_FILENO`, `STDOUT_FILENO` e `STDERR_FILENO` per questi 3 descrittori, di modo che, se nel futuro questi 3 identificatori dovessero assumere altri valori, i vecchi programmi potrebbero essere ancora usati senza modifiche, dopo una ricompilazione.

La funzione `open()` apre un file:

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

int open(const char *pathname, int oflag, ...);
```

Il parametro `oflag` stabilisce la modalità di apertura del file ed è l'OR tra i valori elencati in tabella 3.1. Vale la pena di fare alcune considerazioni:

- se il file è aperto in append, un'operazione di scrittura comporta sempre lo spostamento alla fine del file prima della scrittura.
- se il file è aperto con il flag `O_SYNC`, ogni operazione di write si blocca fino a quando i dati non sono stati effettivamente scritti su disco. Questo è particolarmente utile quando si programmano databases, e si vuole essere sicuri che l'operazione di write sia andata a buon fine.

Per leggere e scrivere da un file si usano le primitive `read` e `write`:

```
#include <unistd.h>

ssize_t read(int filedes, void *buff, size_t nbytes);
ssize_t write(int filedes, const void *buff, size_t nbytes);
```

Entrambe ritornano il numero effettivo di bytes letti/scritti. Infine, la funzione `lseek()` consente di spostare il puntatore corrente del file in una posizione qualunque:

Macro	Modalità
SEEK_SET	spostati a offset bytes dall'inizio del file
SEEK_CUR	spostati a offset bytes dalla posizione corrente
SEEK_END	spostati a offset bytes dalla posizione finale

Tabella 3.2.: flags per la lseek().

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int filedes, off_t offset, int whence);
```

Il parametro `offset` può assumere come valori che vengono interpretati a seconda del parametro `whence`, come elencato in tabella 3.2; la funziona ritorna la posizione assoluta raggiunta dopo il compimento dell'opearazione.

Per esempio, la seguente istruzione:

```
currpos = lseek(fd,0,SEEK_END);
```

sposta il puntatore al file indicato dal descrittore `fd` alla fine.

3.4. Strutture dati coinvolte

Per meglio comprendere cosa succede all'interno del sistema operativo quando viene effettuata una operazione su un file, bisogna analizzare le strutture dati sono coinvolte. Innanzitutto, ogni processo ha una propria tabella dei descrittori di file: un esempio è mostrato in figura 3.2, dove il processo A ha aperto 2 file e il processo B ha aperto un solo file.

Ogni volta che un processo esegue una `open()` viene creata un'entrata nella tabella dei file e una entrata nella tabella dei descrittori del processo. Notare che informazioni come la modalità di apertura (read/write, append, ecc.) e l'offset corrente sono memorizzate nell'entrata corrispondente della tabella dei file e non nella tabella dei descrittori. Inoltre, le informazioni sui file sono contenute direttamente nel file (struttura v-node in figura).

Quindi, ad esempio, se il file è stato aperto con il seguente comando:

```
fd = open("testo.txt", O_WRONLY | O_APPEND);
```

allora una operazione di `write()` sul descrittore `fd` esegue (in maniera atomica) i seguenti passi:

- pone il campo curr offset nell'entrata corrispondente nella tabella dei file pari a current size;
- scrive sul file.

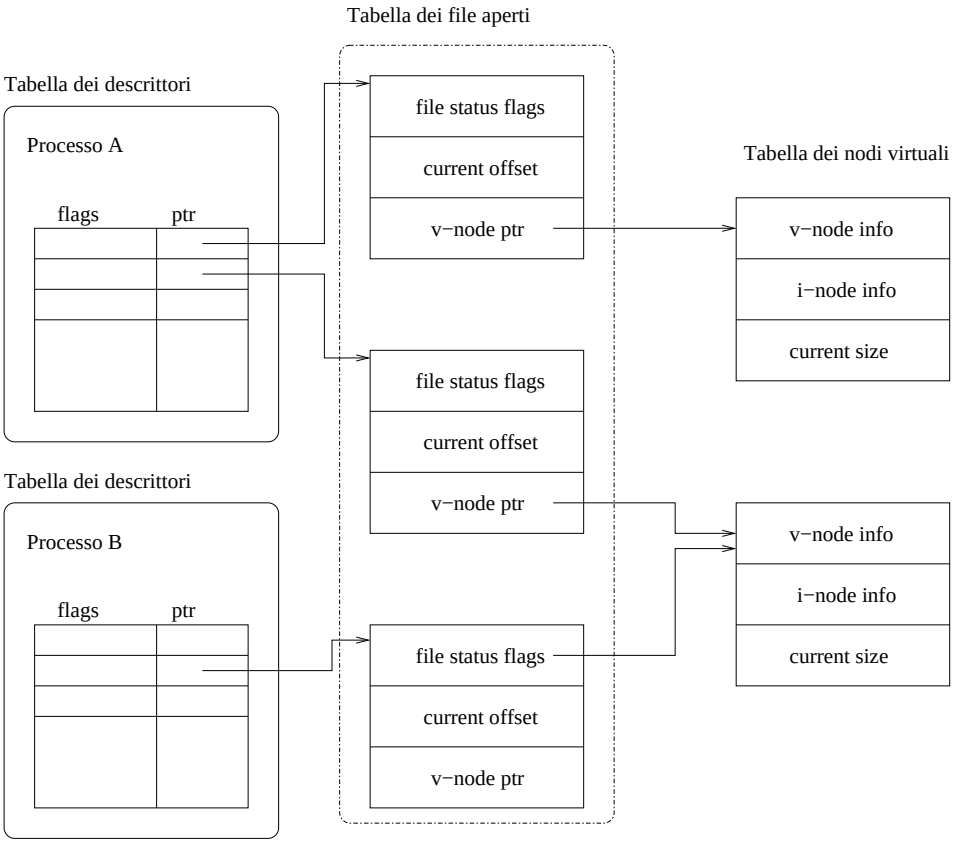

Figura 3.2.: Strutture dati nel kernel.

Domanda Dire se le due sequenze di istruzioni qui di seguito sono equivalenti e perché:

Sequenza 1

```
fd = open("testo.txt", O_WRONLY);
...
lseek(fd, 0, SEEK_END);
write(fd, buff, 100);
```

Sequenza 2

```
fd = open("testo.txt", O_WRONLY | O_APPEND);}
...
write(fd, buff,100);
```

3.4.1. Funzione dup()

Le funzioni `dup()` e `dup2()` servono a duplicare descrittori di file:

```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes1, int filedes2);
```

La `dup()` prende il primo descrittore libero, ci copia dentro il descrittore `filedes` e lo ritorna. La funzione `dup2()` copia il descrittore filedes2 sul descrittore `filedes1`: se filedes1 era un descrittore aperto, lo chiude prima. Le funzioni dup() vengono utilizzate principalmente per redirigere lo stdout e lo stdin di un processo su un file. Vedremo nel prossimo capitolo come la shell implementa le operazioni di re-direzione dell'input e dell'output di un processo tramite queste due funzioni.

3.5. Permessi e proprietà di un file

Ad ogni utente viene associato un ID unico nel sistema (user ID) e almeno un ID di gruppo. Un processo lanciato da un certo utente ha almeno 6 ID:

- **real user ID** e **real group ID** corrispondono alle entrate nel passwd di chi ha lanciato il processo.
- **effective user ID** e **effective group ID** sono gli ID *acquisibili* dal processo con un meccanismo che spiegheremo fra poco. Sono usati per controllare i permessi di accesso ai file.

```
struct stat {
  mode_t st_mode;     /* file type + permission bits */
  ...
  uid_t  stuid;       /* user ID */
  gid_t  st_gid;      /* group ID*/
  off_t  st_size;     /* size (in bytes) */
  time_t st_atime;    /* last access time */
  time_t st_utime;    /* last modif. time */
  time_t st_ctime;    /* last status change time */
};
```

Figura 3.3.: La struttura `struct stat`.

- **supplementary group IDs** sono alcuni gruppi supplementari a cui l'utente può appartenere.
- **saved set–user–ID** e **saved set–group–ID**, vengono salvati da exec, noi non li prenderemo in considerazione.

Ogni file ha un proprietario (*owner*) e un gruppo di appartenenza (*group owner*). Di solito, effective user ID coincide con real user ID (cioè con l'ID di chi ha lanciato il processo) e effective group ID coincide con real group ID. Quindi, se l'utente 150, appartenente al gruppo 300 lancia il comando `ls`, allora questo processo avrà effective user ID pari a 150 e effective group ID pari a 300, non importa chi sia il proprietario del file `/bin/ls`

Esiste però un bit nel descrittore del file eseguibile (detto *set–user–ID*) che, nel caso sia settato, fa in modo che effective user ID coincida con il proprietario del file e effective owner ID coincida con il gruppo di appartenenza del file. Per esempio, il comando `passwd`, che appartiene al root, ha il bit *set-user-ID* settato, e quindi quando viene eseguito ha effective user ID pari all'ID del root. Questo viene fatto per consentire ad un utente normale di poter cambiare la sua password autonomamente: infatti, normalmente un utente non può cambiare il file passwd, e solo il root può farlo.

Ovviamente, l'uso di tale bit è molto pericoloso dal punto di vista della sicurezza e deve essere limitato il più possibile a pochi casi specifici.

3.5.1. File permission bits

Ogni file ha anche un byte nel suo descrittore che specifica i permessi di accesso ai vari utenti/processi del sistema. Tale byte è contenuto nella struttura `struct stat` mostrata in figura 3.3.

Il campo `st_mode` contiene, nei vari bit, il tipo del file e i permessi di accesso al file. I permessi di accesso sono l'OR aritmetico delle seguenti costanti:

S_IRUSR, S_IWUSR, S_IXUSR rispettivamente permessi di lettura, scrittura ed esecuzione per il proprietario del file;

```
drwxrwxr-x   10 lipari   lipari   4096    Nov 1   14:16  compiti
-rw-rw-r--   1  lipari   lipari   16657   Oct 19  15:28  corso.png
drwxr-xr-x   3  lipari   lipari   4096    Oct 29  17:46  dispense
-rw-rw-r--   1  lipari   lipari   306631  Oct 29  17:46  dispense.ps
-rw-rw-r--   1  lipari   lipari   77133   Oct 29  17:46  dispense.tgz
-rw-rw-r--   1  lipari   lipari   4256    Nov 14  11:29  index.html
drwxrwxr-x   7  lipari   lipari   4096    Nov 14  15:16  lezioni
-rw-rw-r--   1  lipari   lipari   86856   Nov 20  11:13  lezioni.tgz
-rw-rw-r--   1  lipari   lipari   4542    Oct 19  15:28  tesi.html
-rw-rw-r--   1  lipari   lipari   2005    Oct 19  15:28  tesi.png
-rw-rw-r--   1  lipari   lipari   3530    Oct 19  15:28  unipi.gif
```

Figura 3.4.: Esempio di output del comando `ls -l`.

S_IRGRP, S_IWGRP, S_IXGRP rispettivamente permessi di lettura, scrittura ed esecuzione per gli utenti appartenenti al gruppo di appartenenza del file.

S_IROTH, S_IWOTH, S_IXOTH rispettivamente permessi di lettura, scrittura ed esecuzione per tutti gli altri utenti del sistema.

Tramite il comando `ls -l` è possibile vedere i permessi di accesso ai vari file, come mostrato in figura 3.4. In particolare, la prima colonna a destra riassume lo stato dei bit del campo `st_mode`:

- il primo carattere indica il tipo del file (`d` sta per directory, `-` sta per un file normale);
- i successivi caratteri, a gruppi di tre, indicano lo stato dei permessi. I primi 3 indicano i permessi di accesso per il proprietario, i secondi 3 i permessi di accesso per un utente dello stesso gruppo, gli ultimi 3 per tutti gli altri.

Quindi ad esempio, il file `unipi.gif` è accessibile in lettura e scrittura sia dal proprietario che dagli utenti del gruppo, mentre è accessibile in sola lettura da tutti gli altri; la directory `dispense` è accessibile in lettura ed esecuzione da tutti, mentre è accessibile in scrittura solo dal proprietario o da un utente dello stesso gruppo.

Le regole di accesso ad un file sono quindi le seguenti:

- Se vogliamo aprire un file (in lettura o in scrittura) dobbiamo avere i permessi di esecuzione sulla directory in cui è contenuto il file e su tutte le directory nel path.
- Se vogliamo creare un file, dobbiamo avere i permessi in scrittura ed esecuzione nella directory.

- Se vogliamo cancellare un file non c'è bisogno di avere il permesso di scrittura sul file, ma basta il permesso in scrittura ed esecuzione sulla directory.

Per cambiare proprietario, gruppo e permessi a un file possiamo utilizzare le funzioni `chown()` e `chmod()`. Da shell, esistono i comandi `chmod`, `chown` e `chgrp`.

```
int chown(const char *pathname, uid_t owner, gid_t group);
int chmod(const char *pathname, mode_t mode);
```

3.6. La I/O Standard Library

Il comitato di standardizzazione ANSI ha formalizzato uno standard per fare I/O bufferizzato che va sotto il nome di I/O Standard Library (`stdio`). Esso è (o dovrebbe essere) indipendente dalla piattaforma su cui è inplementato, quindi potrete ritrovare la stessa libreria sui vari UNIX e sui sistemi Windows. Questa libreria fornisce l'accesso a delle strutture che tentano di facilitare e di ottimizzare l'I/O.

Le funzioni della standard library utilizzano tutte una struttura chiamata `FILE`. L'accesso a questa struttura avviene sempre tramite puntatore e utilizzando le funzioni appropriate: l'utente non dovrebbe mai accedere direttamente ai suoi campi. Un tale tipo di dati viene spesso chiamato "tipo opaco".

Per aprire un file si utilizza la funzione `fopen()`:

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char *type);
```

Il parametro `type` può essere `''r''` (se il vogliamo aprire in lettura), `''w''` (se vogliamo aprire in scrittura) o `''a''` (se vogliamo aprire in append). Se c'è un `+` nella stringa vuol dire che il file viene aperto in read/write. Quindi "`r+`" è equivalente a `''w+''`. Possiamo anche specificare una `''b''` per significare che il file in questione è binario.

> *Su UNIX non c'è differenza fra file testo e file binario. Invece su DOS/Win c'è una bella differenza! Infatti, nei file testo, il carattere di ritorno carrello (CR) su UNIX viene sostituito da due caratteri su DOS/Win (CR + LF). Quindi, quando si trasferisce un file da un sistema DOS/Win a un sistema Unix e viceversa, bisogna stare attenti al tipo di file. Il flag `''b''` nei sistemi UNIX non serve a niente, ma si usa per compatibilità con i sistemi DOS.*

La funzione `fopen()` restituisce un puntatore a una struttura FILE che può essere utilizzato nelle successive chiamate.

Tutte le funzioni della `stdio` realizzano un input/output bufferizzato: nella struttura FILE vi è allocato un buffer di una certa dimensione (BUFSIZ). La prima volta che il programma legge un carattere tramite una delle funzioni della `stdio`, in realtà viene letto un numero di caratteri pari alla dimensione del buffer: successive letture di caratteri si tradurranno in letture dal buffer. Stessa cosa avviene quando un programma scrive un carattere: per prima cosa viene scritto nel buffer: quando il buffer è effettivamente pieno

(oppure in seguito ad un'operazione di flush) il suo contenuto viene effettivamente scritto sul disco.

Questo meccanismo cerca di minimizzare il numero di accessi al mezzo fisico che sono di solito lenti e costosi, cercando di leggere/scrivere i dati "a blocchi".

Un file può essere bufferizzato in 3 modi:

FULLY BUFFERED : il trasferimento dati avviene di solito a blocchi della dimensione del buffer (a meno che non vengano effettuate delle esplicite operazioni di svuotamento).

LINE BUFFERED : il trasferimento dati avviene a "linee", ovvero viene letta/scritta una linea di testo alla volta (la linea finisce con CR).

UNBUFFERED : ogni trasferimento dati da/verso il buffer si traduce in un trasferimento dati da/verso il disco.

I 3 file stdin, stdout e stderr sono rispettivamente line-buffered, line-buffered e unbuffered, mentre un file normale aperto in lettura o scrittura è di solito fully buffered.

Come anticipato, è possibile forzare lo "svuotamento" di un buffer di un file in scrittura tramite la funzione fflush:

```
#include <stdio.h>

int fflush(FILE *fp);
```

Le seguenti funzioni servono per leggere un carattere per volta:

```
#include <stdio.h>

int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

La prima e l'ultima sono in realtà delle macro (per questioni di efficienza). In caso venga raggiunta la fine del file, viene ritornata la costante EOF. In molti sistemi EOF viene definita come -1 che è anche il codice di errore. Per distinguere fra le due situazioni, si utilizzano le due funzioni feof() e ferror(), dal significato abbastanza ovvio.

```
#include <stdio.h>

int ferror(FILE *fp);
int feof(FILE *fp);
```

Le seguenti funzioni fanno l'output a carattere.

```
#include <stdio.h>

int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

Ovviamente, è possibile fare I/O a linee:

```
#include <stdio.h>

char *fgets(char *buf, int n, FILE *fp);
char *gets(char *buf);
int   fputs(const char *str, FILE *fp);
int   puts(const char *str);
```

E infine è possibile fare I/O a blocchi:

```
#include <stdio.h>

size_t fread(void *buf, size_t size, size_t nobj, FILE *fp);
size_t fwrite(void *buf, size_t size, size_t nobj, FILE *fp);
```

Notare che l'utilizzo di una di queste funzioni non ha niente a che fare con il tipo di bufferizzazione del file. Per esempio è possibile utilizzare la funzione `fgetc()` con un file line–buffered e la funzione `fwrite()` con un file unbuffered: sarà la libreria a occuparsi di tradurre le chiamate di funzioni in eventuali trasferimenti da/verso il disco, a seconda del tipo di bufferizzazione del file.

Infine, riportiamo le funzioni per fare input/output formattato di stringhe:

```
#include <stdio.h>

int printf(const char * format, ...);
int fprintf(FILE *fp, const char * format, ...);
int sprintf(char *str, const char * format, ...);

int scanf(const char * format, ...);
int fscanf(FILE *fp, const char * format, ...);
int sscanf(char *str, const char * format, ...);
```

3.6.1. Relazioni tra libreria Posix e libreria ANSI

Dato un puntatore a `FILE` è possibile ottenere il relativo descrittore di file, e viceversa:

```
int   fileno(FILE *fp);
FILE *fdopen(int fd);
```

Va detto che, mentre le funzioni standard POSIX per l'accesso ai file sono primitive di sitema operativo, e quindi atomiche, le funzioni di libreria ANSI sono semplici funzioni di libreria implementate nello spazio di memoria del processo chiamante, e quindi possono essere interrotte a metà. Inoltre, alcune di loro sono *non rientranti*, cioè utilizzano delle strutture globali (ad esempio, il buffer contenuto nella struttura FILE).

Questo può porre qualche problema: per esempio è bene non utilizzare tali funzioni dentro l'handler di un segnale! Supponiamo ad esempio di usare una `printf()` dentro l'handler di un segnale e dentro il programma: se il segnale interrompe l'esecuzione della printf(), il risultato è impredicibile! Infatti, alcune delle strutture globali utilizzate

dalla `printf()` potrebbero essere in uno stato inconsistente quando la printf viene richiamata dall'interno del signal handler. Per maggiori dettagli, vedere il capitolo sulla sincronizzazione tra processi.

4. Multiprocessing

Unix è un sistema operativo nato pensando ad un paradigma di programmazione a scambio di messaggi, in cui le varie attività parallele non condividono spazi di memoria in comune. La multiprogrammazione in ambiente Unix è quindi tradizionalmente basata sul concetto di processo, anche se successive estensioni hanno introdotto concetti nuovi come le zone di memoria condivisa o i processi multithread.

In ogni caso, se non altro per motivazioni di ordine storico, andiamo ad analizzare la programmazione concorrente in ambiente Unix cominciando proprio dalla programmazione multiprocesso.

4.1. Processi

Un processo è in Unix un flusso di esecuzione che può essere eseguito in maniera concorrente e caratterizzato da :

un corpo – Ossia il codice che viene eseguito

uno spazio di memoria privato – Composto a sua volta da :

una zona dati – Suddivisa in dati non inizializzati, o BSS, e dati inizializzati.

uno stack

uno heap – Ossia una zona di memoria in cui il processo può allocare memoria dinamica, tramite, per esempio, la funzione `malloc`.

una tabella di descrittori di file

un descrittore di processo – Il Process IDentificator, o più brevemente PID, è tipicamente un numero che identifica univocamente il processo all'interno del sistema.

uno stato privato

4.1.1. Identificatori di processo

Ogni processo Unix è caratterizzato da un PID privato che serve ad identificare univocamente tale processo all'interno del sistema. Tale PID è *unico* (non possono esistere due processi con lo stesso PID); in pratica si può pensare al PID come ad un "nome" che il sistema assegna ad un processo alla sua creazione e che verrà utilizzato in seguito per riferirsi ad esso.

Ogni processo è in grado di conoscere il proprio PID tramite la syscall `getpid()`, che non prende parametri e restituisce il PID del processo chiamante (è da notare che `getpid()` non può fallire per nessun motivo, quindi non ritornerà mai un valore negativo).

Tramite la syscall `getppid()` è invece possibile ottenere il PID del processo che ha creato il processo chiamante (processo *padre*); se il processo padre è terminato la `getppid()` ritornerà 1, che è per convenzione il PID del processo di sistema `init` (il primo processo ad essere creato). Vedremo nella Sezione 4.1.3 il perché di questo comportamento.

4.1.2. Creazione di nuovi processi

L'unico modo per creare un nuovo processo in UNIX è tramite la primitiva `fork()`.

```
#include <unistd.h>

pid_t fork(void);
```

Questa primitiva crea un processo *figlio* (child) che ha *lo stesso identico codice* del processo che invoca la primitiva, che viene perciò chiamato anche processo *genitore* (parent). Subito dopo l'esecuzione della primitiva `fork()`, dunque, esistono due processi, ognuno con il suo spazio di memoria privato. Sia padre che figlio (in caso di corretta terminazione della syscall) continueranno la loro esecuzione da dopo la `fork()`: tale syscall verrà quindi chiamata una volta e ritornerà due volte (una nel processo padre ed una nel processo figlio).

Il processo figlio è una copia quasi identica del processo padre. In particolare, la `fork()` esegue le seguenti operazioni:

1. crea un nuovo spazio di memoria privato destinato a contenere il processo figlio;
2. crea un nuovo descrittore di processo all'interno del nucleo del sistem operativo;
3. assegna un nuovo PID al processo figlio (infatti, come detto, il PID deve identificare univocamente un nuovo processo);
4. esegue una copia *quasi* fedele della memoria del padre nella memoria del figlio, sia per quanto riguarda la parte dei dati, che per quanto riguarda il codice;
5. a seconda della politica di schedulazione del sistema operativo, uno dei due processi, il padre o il figlio, andrà in esecuzione, mentre l'altro resterà in attesa di essere eseguito dal processore; nei sistemi con multiprocessori, entrambi i processi possono andare in esecuzione contemporaneamente su due processori diversi;
6. ritorna due volte: nel processo padre, la primitiva `fork()` ritorna il valore del PID assegnato al processo figlio; nel processo figlio, ritorna 0.

Il punto 4 è molto importante e merita di essere approfondito meglio con un esempio. Se il processo padre ha dichiarato una variabile intera `a` il cui valore subito prima di eseguire la `fork()` e' di 5, subito dopo la `fork()` sia il padre che il figlio possiedono una variabile `a = 5`. Però, da questo momento in poi, le due variabili sono completamente distinte, una sta nello spazio privato del processo padre e l'altra sta nello spazio privato del processo figlio, e avranno vita diversa. Quindi, se successivamente il processo padre esegue l'istruzione `a++`, solo la sua variabile `a` verrà incrementata, mentre la variabile `a` che sta nel processo figlio conserverà il valore originale.

I due processi possono essere distinti consultando il valore di ritorno della `fork()`: il padre otterrà il PID del processo figlio, mentre il figlio otterrà 0. Un utilizzo della `fork()` può quindi essere il seguente:

```
pid_t res;

int a = 1;
b = 2;

res = fork();
if (res < 0) {
  perror("fork failed");
  exit(-1);
}

a++; // 1: Codice comune: sia il padre che il figlio eseguono
     //    questo pezzo di codice!
b--; // 1:

if (res == 0) {
  // 2: Solo il figlio esegue questo codice
  a++;
  ...
}
else {
  //  3: solo il padre esegue questo codice
  b = 5;
}
// 4: sia il padre che il figlio possono potenzialmente
// eseguire questo codice
```

Prima della `fork()` esiste un solo processo, che dichiara 3 variabili, `res`, `a` e `b`. Subito dopo la `fork()` esistono 2 processi, ognuno con il proprio spazio di memoria privato. Il processo padre possiede ancora 3 variabili, il cui valore è `res =` *pid del figlio*, `a = 1` e `b = 2`. Il processo figlio possiede anch'egli le stesse 3 variabili, con valori `res = 0`, `a = 1` e `b = 2`.

La parte di codice indicata con *1:* viene eseguita de entrambi i processi: quindi, sia il padre che il figlio incrementano le loro rispettive variabili a e decrementano le loro variabili b, che quindi continuano ad avere gli stessi valori.

La parte di codice indicata con *2:* viene eseguita esclusivamente dal processo figlio: infatti, soltanto nel processo figlio la variabile `res` ha il valore 0. Quindi, il processo figlio incrementa la sua variabile `a` che assume quindi il valore 3, mentre la variabile `a` del padre rimane al valore 2.

La parte di codice indicata con *3:* viene eseguita esclusivamente dal processo padre, e quindi la sua variabile `b` assumerà il valore 5, mentre la variabile `b` del figlio resterà al valore 1.

Infine, la parte di codice indicata con *4:* potrebbe, in generale, essere eseguita da entrambi i processi, a meno che uno dei due processi non decida di terminare prima nel suo ramo dell'if. Ad esempio, potrebbe darsi che il processo figlio invochi la primitiva `exit` nella parte *1:*, e quindi non raggiungerà mai la parte *4:*.

Come si vede, una soluzione di questo tipo non è molto pratica, in quanto obbliga a scrivere il codice del processo figlio e quello del processo padre nei due blocchi `then` e `else` di un'istruzione `if`. Una soluzione per rendere il codice più leggibile può essere la seguente:

```
pid_t = res;
...
res = fork();
if (res < 0) {
  perror("fork failed");
  exit(-1);
}
if (res == 0) {
  /* We are in the child process */
  corpo_figlio();
  exit(1);
}
/* We are in the father process */
...
```

In questo caso, il corpo del figlio è stato spostato in una funzione separata, il maniera da rendere meno confuso il codice. Naturalmente, ci sono decine di modi di strutturare un programma, la fantasia è il solo limite!

Esercizio Scrivere un programma in cui il padre crea 5 processi figli, ognuno dei quali stampa a video il proprio PID e poi esce.

4.1.3. Terminazione dei processi

Abbiamo visto come sia possibile creare un nuovo processo tramite la syscall `fork`, occupiamoci ora della terminazione dei processi.

Come spiegato nel capitolo 2, un processo può terminare esplicitamente tramite la syscall `_exit()`, o la library call `exit()`, o implicitamente quando l'esecuzione del main termina (ciò può avvenire perché il flusso di esecuzione è arrivato alla fine della funzione `main()` o perché essa ha effettuato una `return`).

La corretta terminazione di un processo presuppone che esso ritorni un valore tramite la funzione `exit()` o l'istruzione `return`; tale valore può essere recuperato dal processo padre per determinare l'esito dell'esecuzione del figlio. Per fare questo, il padre ha bisogno di sincronizzarsi col figlio, attendendo la sua terminazione tramite una primitiva della famigla `wait()`. La più semplice di tali funzioni è la `wait()`, che permette ad un processo di attendere la terminazione di uno dei suoi processi figli, recuperando il valore di uscita ritornato dalla funzione `exit()`. Ecco quindi come funziona la `wait()`:

- se il processo chiamante non ha figli, la syscall fallisce, ritornando un codice di errore < 0;
- se il processo chiamante ha creato dei processi figli con `fork()`, ma nessuno di essi è ancora terminato, la syscall si blocca;
- se il processo chiamante ha dei figli che sono già terminati, ritorna il PID ed il valore di uscita del primo processo figlio terminato.

Il prototipo della syscall è il seguente:

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

pid_t wait(int *status);
```

Argomenti in ingresso :

status – Puntatore ad un intero in cui registrare lo stato di iscita del processo figlio.

Valore restituito :

-1 – Se il processo non ha figli

pid – Il PID di un processo figlio terminato.

Un esempio tipico di utilizzo della `wait()` è il seguente:

```
res = wait(\&status); /* This syscall can block */

if (res < 0) {
  /* No children to wait for */
  perror("Wait failed");
  exit(-1);
}
/* Now the value returned from the child is in status */
```

Il valore di ritorno di un processo in generale risponde ad alcune convenzioni di base: come detto, un valore negativo è segno di una terminazione anomala. Per poter analizzare tali valori di ritorno, il sistema mette a disposizione alcune macro: `WIFEXITED`, `WIFSIGNALED` e `WIFSTOPPED`. La macro `WIFEXITED` applicata al valore `status` ritorna 1 se il processo figlio è terminato correttamente (ed in questo caso, `WEXITSTATUS` ottiene il

valore >= 0 ritornato dal figlio). Le macro `WIFSIGNALED` e `WIFSTOPPED` indicano invece se il processo figlio è stato interrotto da un segnale o è stato bloccato; in questo caso, le macro `WTERMSIG` e `WSTOPSIG` permettono di ottenere il numero del segnale che ha terminato o bloccato il processo (vedi Sezione 5.1. In alcuni sistemi è anche definita la macro `WSTOPSIG`, per vedere se è avvenuto un *core dump*, con conseguente generazione del file `core`. Si invita a consultare le man page per avere maggiori informazioni su tali macro e su altre eventualmente definite.

Per permettere la sincronizzazione ed il passaggio del valore di ritorno come specificato precedentemente, il kernel deve mantenere informazioni relative ai processi anche dopo la loro terminazione, finché il loro processo padre non effettua una `wait()`. Questi processi inattivi che continuano ad occupare posto nella tabella dei processi, detti *processi zombie*, possono causare un inutile spreco di descrittori dei processi. Per tale motivo, è sempre bene effettuare una `wait()` su tutti i processi figli creati.

Quando un processo termina, tutti i suoi processi figli (compresi quelli che sono nello stato di *zombie*) vengono ereditati dal processo di sistema `init` (in pratica, init diventa automagicamente il padre di tutti i processi il cui padre termina). Questo meccanismo permette di evitare la creazione di un eccessivo numero di processi *zombie*, in quanto quando un processo figlio di `init` termina, `init` effettua automaticamente una `wait` (nella Sezione 5.1, parlando dei segnali, vedremo come è possibile fare ciò).

Se nessuno dei figli è terminato, la `wait()` ha un comportamento bloccante; esistono invece altre syscall che prevedono semantiche non bloccanti. In particolare, la syscall `waitpid()` permette di specificare di attendere la terminazione un determinato processo, ed inoltre accetta in input un flag che specifica un comportamento non bloccante. Altre due syscall, `wait3()` e `wait4()`, derivanti dai sistemi BSD, sono sostanzialmente analoghe, cambiando leggermente la sintassi. Si raccomanda di riferirsi alle manpage per avere informazioni su sintassi e semantica di `waitpid()`, `wait3()` e `wait4()`.

4.1.4. Caricamento di un programma in memoria

Per evitare di dover scrivere il corpo dei due processi nello stesso file, Unix mette a disposizione la famiglia di funzioni `exec()`.

Come detto, la primitiva `fork()` permette di creare un nuovo processo, copiando nel suo spazio di memoria lo spazio di memoria del processo creante, e questo obbliga a inserire il codice ed i dati di tutti i processi nello stesso eseguibile, con conseguente aumento delle dimensioni degli eseguibili, complicazione della struttura del programma e perdita di flessibilità.

Inoltre, si pone il problema di eseguire un processo esterno già esistente, scritto da un altro utente. Ad esempio, supponiamo di voler ottenere informazioni sui processi attualmente in esecuzione nel sistema. Un metodo abbastanza semplice potrebbe essere quello eseguire il programma `ps`, presente in tutte le distribuzioni UNIX. Questa operazione viene anche chiamata *spawn*: cioè si crea un processo che esegue un programma esterno.

Per permettere questo, Unix mette a disposizione la famiglia di funzioni `exec()`. Onguna delle funzione della famiglia, sebbene con modalità leggermente differenti si occupa

di sovrascrivere i dati, il segmento BSS ed il codice di un processo con un'immagine prelevata da disco.

È da notare che una funzione della famiglia `exec()` non crea un nuovo processo, ma semplicemente modifica lo spazio di memoria del processo che la chiama; per questo motivo, **in caso di corretto funzionamento la `exec()` non ritornerà**. È quindi chiaro che se la syscall ritorna un valore al chiamante, tale valore è necessariamente minore di 0 (tipicamente -1). A livello intuitivo, una `exec()` è paragonabile ad una `jmp` nel codice di un altro file oggetto, che rimpiazza il primo nella memoria privata del processo.

La famiglia di funzioni `exec` è composta da `execl()`, `execlp()`, `execle()`, `execv()`, `execvp()` ed `execve()`; tipicamente, tutte tali funzioni sono basate sulla `execve()`, costituendo dei meri front-end verso di essa. Per questo motivo, in questa sede tratteremo solo la `execve()`, rimandando ai manfile (`man exec`) per la sintassi e la semantica delle altre syscall della famiglia.

```
#include <unistd.h>

int execve (const  char  *filename, char *const argv [],
            char *const envp[])
```

Argomenti in ingresso :

filename – Nome del file da caricare in memoria

argv – Lista dei parametri

argv – Lista delle variabili di ambiente.

Valore restituito :

-1 – In caso di errore

Non ritorna – In caso di corretto funzionamento.

Carica in memoria l'eseguibile indicato dal parametro `filename` (nella directory corrente), passandogli come parametri la lista di stringhe indicata da `argv`, e creandogli un ambiente con definite le variabili specificate da `envp`.

Il parametro `argv` è un array di puntatori a stringhe, ognuna delle quali specifica un parametro passato al programma. L'ultimo elemento dell'array deve essere `NULL` (0). Per convenzione, **il primo parametro deve sempre essere il nome dell'eseguibile**.

Ecco quindi come usare la `execve()`:

```
char = *argv[5];
int res;
...

argv[0] = "ls"
argv[1] = "/home";
argv[2] = NULL;
```

```
  res = execve("/bin/ls", argv, NULL);
  if (res < 0) sys_err("execve failed");

  /* We cannot be here!!! */
  printf("Panico!!!");
  exit(-1);
```

4.1.5. Un esempio

Riportiamo di seguito un esempio funzionante di quanto detto sopra: in pratica, un processo padre forka un figlio e carica un nuovo corpo per il processo figlio prelevandolo da un altro eseguibile. A questo punto il padre attende un secondo e poi termina, mentre il figlio non fa altro che notificare la sua esistenza indicando i parametri ricevuti in ingresso.

```
/* Esempio di exec ....
 * Quando un processo figlio finisce al padre viene   *
 * inviata la SIGCHLD. In questo esempio viene quindi *
 * mostrato anche l'uso della SIGCHLD ....            */

/* Quesito : Nel caso in cui l'output venga rediretto *
 * su file, la prima printf del processo padre puo'   *
 * stampare lo stesso messaggio per due volte. Se     *
 * l'output invece viene stampato su video, questo    *
 * non avviene mani. Sapete spiegare il perche' ?     */

#include <sys/types.h>
#define __USE_POSIX
#include <stdio.h>
#include <unistd.h>
#include "myutils.h"

int main (void)
{
  pid_t child;
  /* Array usato nell'esempio per passare i parametri  *
   * al task generato tramite una exec                 */

  char *array[3] = {"arg1", "arg2", "arg3"};

  if ((child = fork()) < 0) sys_err("Errore nella fork");
  else if (child == 0) {
    /* A questo punto sono nel figlio */
    printf("Processo figlio ... padre = \%d\n"}, getppid());

  /* Scommentare la linea seguente e commentare quella    *
   * ancora dopo per vedere le differenze far i due tipi  *
   * di passaggio di parametri alla exec.                 */
```

```
35    /* if (execlp("figlio", "figlio", "arg1", "arg2") < 0) { */
      if (execvp("figlio", array) < 0)
37      sys_err("E' fallita la exec");
      else {
39      /* se va a buon fine, la exec non ritorna! */
        /* Sono nel processo padre                 */
41      printf("Processo padre .... pid_t = \%d\n", getpid());
        sleep(1);
43    }
      return 0;
45  }
```

```
1   /* Programma che viene eseguito tramite la primitiva   *
     * exec. Si limita a stampare a video i parametro che  *
3    * gli sono passati in input                           */

5   #include <stdio.h>
    #include <unistd.h>
7
    int main(int argc, *argv[])
9   {
      int i;
11
      printf("Programma 'execed' - Processo padre = \%d\n",
13           getppid());
      for (i=0; i<argc; i++)
15      printf("Argomento %d = %s \n", i, argv[i]);

17    return 0;
    }
```

4.2. Esercizi

Esercizio 1 – Scrivere un programma che prende un numero $N > 1$ dalla riga di comando: quindi forka un figlio, il quale a sua volta forka un altro figlio, e così via, fino ad ottenere N processi in tutto.

Esercizio 2 – Come l'esercizio 1, con la differenza che ogni processo, prima di terminare, aspetta la terminazione del proprio figlio. L'ultimo processo creato, naturalmente, non aspetta niente!

Esercizio 3 – Come l'esercizio 1, soltanto che ogni figlio, prima di uscire, aspetta che sia terminato il proprio padre.

5. Sincronizzazione fra processi

Abbiamo visto come creare e terminare processi, come caricare il corpo e i dati di un processo da un'immagine su disco e come sincronizzare un processo con la terminazione dei suoi figli recuperando i valori da essi ritornati. In generale, i processi che compongono un programma concorrente devono però sincronizzarsi e comunicare tra loro in maniera non banale. Inoltre, a volte il kernel deve comunicare a un processo che una certa condizione (eccezionale o no) è avvenuta.

In questo capitolo ci occuperemo di segnali, ovvero di comunicazioni asincrone che segnalano una certa condizione. I segnali possono essere utilizzati sia per comunicare condizioni di errore o eccezioni, sia per sincronizzare due processi fra loro.

5.1. I segnali

I *segnali* sono il meccanismo utilizzato dai sistemi Unix per rendere possibile la comunicazione asincrona di certe condizioni. In pratica, un segnale è un evento asincrono (talvolta designato col nome di interrupt software anche se impropriamente), che un processo può *inviare* ad un altro processo (o ad un gruppo di processi). Il meccanismo è in realtà molto generale, in quanto anche il kernel è in grado di generare segnali da mandare ai vari processi per segnalare condizioni di errore o eccezioni. Il destinatario può reagire in maniera diversa all'arrivo di tale evento:

- ignorare il segnale,
- eseguire un *signal handler* specificato dal processo,
- uscire (abortendo).

Un segnale che non ha ancora generato nessuna delle tre azioni sopra è detto *pendente*.

Nelle prossime sezioni saranno introdotte alcune primitive di nucleo per permettere a un processo di:

- specificare l'azione da intraprendere in risposta ad un segnale, (*installare un signal handler*);
- bloccare un segnale (*mascherarlo*);
- controllare se ci sono segnali pendenti;
- inviare segnali ad un altro processo.

Purtroppo, la semantica di tali funzioni (e la semantica dei segnali stessi) dipende fortemente dal sistema, per cui si rimanda alle man page di ogni specifica implementazione di Unix per una descrizione precisa. In questa sede, ci limiteremo a spiegare la sintassi e la semantica delle funzioni previste dallo standard POSIX. In particolare, ancora una volta, gli esempi presentati sono stati provati su un sistema Linux.

5.2. Specifica di un signal handler

Ad ogni processo è assegnata una tabella dei segnali, che fa parte del suo stato privato e specifica l'azione che il processo deve eseguire in risposta ad un segnale. Al momento della creazione del processo, ad ogni segnale è assegnato un comportamento di default, che può essere cambiato tramite un'apposita system call. Storicamente, i sistemi Unix prevedono la funzione `signal()`, che permette di assegnare un signal handler (tipicamente, una funzione del tipo `void f(int s)`) al segnale specificato e ritorna il signal handler precedentemente assegnato al segnale, sempre sotto forma di un puntatore a una funzione.

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

Argomenti in ingresso :

signum – Identificativo della signal di cui si vuole installare l'handler.

handler – Puntatore alla funzione che contiene il codice dell'handler, oppure una delle due macro predefinite : SIG_IGN, che ha il significato di ignorare la signal specificata; SIG_DFL, che installa l'handler di default della signal specificata.

Valore restituito :

SIG_ERR – In caso di errore.

old_signal – Puntatore alla precedente signal.

Di seguito riportiamo un piccolo esempio di come settare un nuovo handler per la signal di identificativo `SIGUSR1` :

```
#include <signal.h>
#include "myutils.h"

/* Forward declaration del corpo dell'handler */
static void handler(signo);

int main(void)
{
  ...
  /* Installo l'handler e controllo il risultato */
  if (signal(SIGUSR1, handler) == SIG_ERR)
```

```
    sys_err("Errore nell'allocazione delle SIGUSR1");
  ...
}
```

Il comportamento di questa syscall non è però standard, in quanto in alcuni sistemi il signal handler viene resettato al valore di default quando una signal scatta, mentre su altri rimane settato al valore specificato dalla `signal` anche in seguito allo scattare del segnale (Linux permette entrambe le semantiche, semplicemente linkando librerie differenti; al solito, riferirsi alla man page).

Per evitare questo problema, lo standard POSIX definisce una nuova chiamata di sistema, `sigaction`, che è standard e quindi un programma scritto utilizzando tale chiamata è sicuramente più portabile. Qui di seguito riportiamo il prototipo della `sigaction`:

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldct);
```

Argomenti in ingresso :

signum – Identificativo del segnale di cui si vuole installare l'handler.

act – Puntatore alla struttura contenente i parametri che specificano il comportamento del nuovo handler.

oldact – Puntatore alla struttura in cui, se diverso da `NULL`, vengono salvate le informazioni riguardanti il vecchio handler.

Valore restituito :

0 – In caso di successo.

-1 – In caso di errore

Come si nota le descrizioni dei signal handler sono fornite tramite la struttura di tipo `struct sigaction`, così definita:

```
#include <signal.h>

struct sigaction {
  void      (*sa_handler)();
  sigset_t sa_mask;
  int sa_flags;
};
```

Significato dei vari campi :

sa_handler – Puntatore al codice del nuovo handler, oppure `SIG_DFL` se vogliamo il comportamento standard, oppure `SIG_IGN` se vogliamo ignorare il segnale.

sa_mask – Maschera dei segnali che devono essere bloccati durante l'esecuzione dell'handler.

sa_flags – Flags per specificare il comportamento del sistema durante l'esecuzione dell'handler.

Per installare un nuovo handler vanno quindi fatti i seguenti passi:

- scrivere il relativo codice sotto forma di funzione C,
- dichiarare un'istanza di struttura `sigaction` e settarne in maniera opportuna i campi,
- chiamare la syscall opportuna.

Nell'esempio seguente si mostra come installare un handler usando la semantica della `sigaction()`:

```
#include <signal.h>
#include "myutil.h"

/* Forward declaration del corpo dell'handler */
static void handler (int signo);

int main(void)
{
  struct sigaction nuova, vecchia;

  nuova.sa_handler = handler;
  /* Tutte le signal sono mascherate durante
   * l'esecuzione dell'handler */

  sigemptyset(&nuova.sa_mask);
  nuova.sa_flags = 0;

  /* Installo l'handler e controllo il risultato */
  if (sigaction(SIGUSR1, &nuova, &vecchia) == -1)
     sys_err("Errore nell'allocazione delle SIGUSR1 \n");

  ...
}
```

Nell'esempio sopra è introdotta la funzione `sigemptyset()`. Questa fa parte di una famiglia di funzioni che servono per settare correttamente il campo `sa_mask` della struttura `sigaction`. Questo campo serve per specificare il comportamento dell'handler in caso di annidamento dei segnali. Può capitare, infatti, che scatti un segnale mentre il task sta eseguendo l'handler relativo ad un altro. In questo caso il programmatore ha due possibilità:

- interrompere l'esecuzione dell'handler e servire il nuovo segnale con il suo relativo handler; al termine del nuovo handler, il programma ricomincerà a eseguire l'handler precedentemente interrotto.
- ritardare l'esecuzione del nuovo handler fino a che il vecchio non abbia finito. Questa operazione viene anche chiamate mascheramento del segnale.

Si può specificare tale comportamento per ogni handler installato e questo è fatto tramite una famiglia di funzioni di cui la `sigemptyset()` è solo una delle appartanenti. Tali funzioni operano su una struttura di tipo `struct sigset_t` che contiene tutte le informazioni relative ai signal da mascherare e che, come visto, è uno dei campi della `sigaction()`. Per ogni segnale, la `struct sigset_t` prevede un bit che può quindi assumete il valore di 0 o 1.

La struttura `struct sigset_t` è un cosiddetto *tipo opaco*: in altre parole, un programma non deve modificare direttamente il valore dei suoi campi (che il programmatore non dovrebbe neanche conoscere), ma può accedere ad essa solo attraverso un insieme di macro o funzioni che definiscono l'*interfaccia* del tipo. Infatti, in molti sistemi vengono definiti al massimo 31 segnali, quindi la struttura dati può consistere di un solo campo di tipo `int` mentre in altri sistemi possono essere definiti più di 32 segnali, quindi un intero non basta più. Per rendere l'accesso a tale struttura dati quanto più portabile possibile, è possibile modificare il bit relativo ad ogni segnale soltanto tramite le seguenti funzioni:

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Descrizione delle funzioni :

sigemptyset – Modifica la struttura `set` in modo che tutti i bit siano posti a 0.

sigfillset – Modifica la struttura `set` in modo che tutti i signal siano posti a 1.

sigaddset – Aggiunge alla struttura `set` il signal `signum`, ossia pone il bit corrispondente a 1.

sigdelset – Rimuove il signal `signum` dalla struttura `set`, ovvero pone il corrisponente bit a 0.

sigismember – Controlla se il signal `signum` è pari a 1 o a 0 nella struttura `set`.

Valore restituito :

0 – In caso di successo.

-1 – In caso di errore.

La funzione `sigismember()` fa eccezione ed restituisce i seguenti valori:

1 – Se il bit corrispondente al signal `signum` è pari a 1 nella struttura `set`.

0 – Se il bit corrispondente al signal `signum` è pari a 0 nella struttura `set`.

-1 – In caso di errore.

A seconda di come la `struct sigset_t` viene utilizzata, si hanno comportamenti diversi nel sistema. Di solito essa viene utilizzata come *maschera dei segnali*. In questo casi il significato è che tutti i segnali il cui bit è settato a 1 vengono *mascherati* (ovvero vengono bloccati e rimangono pendenti).

Il campo `sa_flags` della struttura `struct sigaction` è usato per specificare comportamenti particolari da seguire quando scatta un segnale per cui l'handler è stato installato. Non ci dilungheremo qui a speigare tutti i possibili valori che tale campo può assumere. Per fare un esempio ci limitiamo a segnalare tre possibili valori di questo campo:

SA_ONESHOT or **SA_RESETHAND**. Quando questo flag viene specificato, subito dopo che il segnale è stato servito dall'handler, il comportamento standard per quel segnale viene rispristinato. Quindi, l'handler può scattare una sola volta, a meno che non si re-installi l'handler successivamente.

SA_NOMASK or **SA_NODEFER**. Normalmente un handler corrispondente al segnale non può essere interrotto da un'altra occorrenza dello stesso segnale. Per esempio, se installiamo un handler per il segnale `SIGUSR1` e il segnale scatta ancora mentre l'handler sta eseguendo, normalmente il segnale viene bloccato fino alla fine dell'handler. Se vogliamo invece che il segnale venga servito immediatamente interrompendo il suo stesso signal handler, allora possiamo specificare `SA_NOMASK` nel campo `sa_flags`.

SA_RESTART . Come vedremo più avanti, le system call bloccanti come ad esempio la `read` vengono interrotte e abortite se arriva un segnal mentre la system call è bloccata. Se vogliamo che la system call continui a rimanere bloccata dopo l'esecuzione del signal handler, dobbiamo specificare `SA_RESTART`.

Come ultimo nota, bisogna specificare che non tutti i segnali possono essere specificati nella `sigaction`. In particolare, se specifichiamo `SIG_IGN` oppure un handler nel campo `sa_handler` per i segnali `SIGKILL` e `SIGSTOP`, la `sigaction` non ha alcun effetto.

5.3. Invio di un segnale

Una volta che abbiamo visto cosa sono i segnali e come definire un handler per essi (o come comunicare al sistema che un task intende ignorare i segnali di un determinato tipo), vediamo come sia possinile generare un segnale ed inviarlo ad un determinato task.

Alcuni segnali sono generati automaticamente dal sistema (in particolare dal kernel) quando si verificano determinate condizioni di errore, mentre altri possono essere generati da un processo ed indirizzati ad un altro processo appartenente allo stesso utente, con scopi di comunicazione, di sincronizzazione, o di altro genere (per esempio, quando un utente preme CTRL-C, la shell invia un segnale `SIGINT` al processo in esecuzione per terminarlo).

La syscall da utilizzare per generare un segnale è la `kill()`, che funziona in modo analogo all'omonimo comando di Unix (in effetti, il comando `kill` utilizza tale syscall). Vediamo allora la sintassi della syscall `kill()`:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

Argomenti in ingresso :

pid – identificatore del processo destinatario del segnale

sig – numero del segnale da inviare.

Valore restituito :

-1 – In caso di errore

0 – In caso di corretto funzionamento.

Invia il segnale identificato da `sig` al processo o ai processi identificati da `pid`. Se `pid` è maggiore di 0, indica il PID del processo a cui il segnale deve essere inviato, se `pid` è uguale a 0, il segnale viene inviato a tutti i processi che stanno nello stesso gruppo del processo mittente, se `pid` vale -1, il segnale è inviato a tutti i processi (tranne il processo numero 1, `init`, che è un processo speciale), mentre se `pid` è minore di -1, allora viene interpretato come un identificatore di gruppo, e tutti i processi appartenenti al gruppo `-pid` riceveranno il segnale.

E' da notare che per ovvi motivi di sicurezza un processo non può inviare un segnale ad un qualsiasi altro processo (altrimenti un utente potrebbe forzatamente terminare i processi di altri utenti), ma in generale può inviare segnali solo a processi appartenenti allo stesso utente. Unica eccezione sono i processi che girano col privilegio di "root" (i processi del superuser), i quali possono inviare segnali a qualsiasi altro processo, tranne il già citato processo `init`. Se si tenta di inviare un segnale ad un processo senza averne il diritto, la syscall `kill()` fallisce, ritornando -1. Ultima cosa da notare è che se si specifica `sig` uguale a 0, la `kill()` non invial alcun segnale, ma i controlli sui permessi vengono ugualmente effettuati: in questo modo è facile verificare se un processo ha i permessi per inviare un segnale ad un altro.

5.4. Sospensione in attesa di un segnale

Un processo può interrompersi in attesa dell'arrivo di un segnale. Un modo per farlo è attraverso la primitiva `pause()`. Se un processo invoca questa primitiva si blocca. Se, dopo essersi bloccato arriva un segnale e tale segnale non è mascherato, il processo si sblocca, un eventuale signal handler viene eseguito, e al ritorno il processo prosegue con l'istruzione successiva alla `pause()`.

```
#include <unistd.h>

int pause(void);
```

Valore restituito :

-1 – In ogni caso viene restituito -1 e la variabile `errno` viene settata al valore `EINTR` per indicare che la primitiva è stata interrota da un segnale.

Un altro modo per sospendersi è tramite la primitiva `sleep()`.

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Argomenti in ingresso :

seconds – numero dei secondi di sospensione;

Valore restituito :

0 – Nel caso in cui il numero di secondi specificato è passato;

n – Nel caso in cui la primitiva sia interrotta da un segnale, viene tornato il numero di secondi rimasti da trascorrere e la variabile `errno` viene settata a `EINTR`.

Le primitive `sleep()` e `pause()` non sempre sono il modo migliore per sospendere un processo, specialmente se vogliamo realizzare delle sincronizzazioni particolari. Il problema viene spiegato dall'esempio seguente.

```
#include <sys/types.h>
#define __USE_POSIX
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include "myutil.h"

int seconds;

static void handler(int signo)
{
  seconds++;
```

```
14 }

16 int main(void)
   {
18   pid_t child, father;
     struct sigaction nuova, vecchia;
20
     seconds = 0;
22
     nuova.sa_handler = handler;
24   sigemptyset(&nuova.sa_mask);
     nuova.sa_flags = 0;
26
     sigaction(SIGUSR1, &nuova, &vecchia); // Installa l'handler
28
     if ((child = fork()) < 0) sys_err("Errore nella fork .... \n");
30   else if (child == 0) {
       father = getppid();
32     for(;;) {
         sleep(1);
34       kill(father, SIGUSR1);
         pause();
36       printf("Secondi trascorsi (figlio) %d -- time = %d\n",
                2*seconds, (int)time(0));
38     }
     } else {
40     if (seconds == 0) pause();
       for(;;) {
42       printf("Secondi trascorsi (padre) %d -- time = %d \n",
                2*seconds-1, (int)time(0));
44       sleep(1);
         kill(child, SIGUSR1);
46       pause();
       }
48   }
   }
```

Il programma crea due processi. Alternativamente ognuno dei due processi invia un signal all'altro e quindi sospende temporaneamente la sua esecuzione, tramite l'uso della funzione `pause()`, in attesa di essere *risvegliato* dalla signal inviata dall'altro.

Il programma appena descritto può andare in deadlock. Supponiamo infatti che succeda la seguente situazione:

- il processo figlio esegue l'istruzione `kill(father, SIGUSR1);` e poi viene preemptato dal sistema operativo *prima di poter eseguire la pause()*.
- il processo padre si sveglia, esegue l'handler e dopo essersi sospeso per un secondo, esegue l'istruzione `kill(child, SIGUSR1);`; quindi si sospende con la `pause()`;

- come effetto della kill, il processo figlio esegue il signal handler e, alla terminazione, si sospende sulla `pause()`, in attesa di un segnale dal padre;
- il padre è bloccato e non può svegliarsi, in quanto aspetta un segnale dal figlio.

Sebbene un caso del genere sia molto raro, è indubbiamente possibile.

Per evitare del tutto il caso appena descritto, bisognerebbe essere in grado di *mascherare* il segnale `SIGUSR1` fino alla sospensione, e poi, *con una singola operazione atomica*, smascherare il segnale e sospendersi.

Per realizzare un tale meccamismo, si usano le due primitive `sigprocmask()` per mascherare le interruzioni e `sigsuspend()` per sospendersi.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigsuspend(const sigset_t *mask);
```

La `sigprocmask()` permette di mascherare alcuni segnali. Il parametro `set` indica la maschera da applicare secondo la modalità specificata dal parametro `how`. Il parametro `how` può assumere 3 valori:

- `SIG_BLOCK` significa che la maschera `set` viene applicata con un or logico sulla maschera corrente; quindi, se un segnale era già bloccato, per esso la chiamata non ha effetto; se invece un segnale non era bloccato e nel parametro `set` il bit corrispondente è a uno, il segnale verrà bloccatto come effetto della chiamata.
- `SIG_UNBLOCK` significa che i segnali specificati in `set` (quindi con il corrispondente bit a 1) vengono sbloccati come effetto della chiamata.
- `SIG_SETMASK` significa che la maschera specificata con `set` diviene la maschera corrente.

La vecchia maschera viene restituita in `oldset`.

La `sigsuspend` fa due cose: applica temporaneamente la maschera specificata da `mask` e sospende il processo in attesa che arrivi un segnale non mascherato in `mask`. Quando il processo si risveglia, la vecchia maschera viene ripristinata subito prima del ritorno della `sigsuspend()`.

L'esempio di prima può essere riscritto come di seguito.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>

void sys_err(char *s)
{
  puts(s);
```

```
  exit(-1);
}

int seconds;
static void handler(int signo)
{
  seconds++;
}

int main()
{
  pid_t child, father;
  struct sigaction nuova, vecchia;
  sigset_t curr, mask1;
  long long i;

  seconds = 0;

  nuova.sa_handler = handler;

  sigemptyset(&nuova.sa_mask);
  nuova.sa_flags = 0;

  // Installa l'handler
  sigaction(SIGUSR1, &nuova, &vecchia);
  // Serve a leggere la maschera corrent in mask1
  sigemptyset(&curr);
  sigprocmask(SIG_BLOCK, &curr, &mask1);
  // Blocca SIGUSR1
  sigaddset(&mask1, SIGUSR1);
  // curr è ora la maschera vecchia
  sigprocmask(SIG_BLOCK, &mask1, &curr);

  // da ora in poi SIGUSR1 è bloccato

  if ((child = fork()) < 0) sys_err("Errore nella fork .... \n");
  else if (child == 0) {
    father = getppid();
    for(;;) {
      sleep(1);
      kill(father, SIGUSR1);
      // mentre il figlio è sospeso,
      // SIGUSR1 è non bloccato
      sigsuspend(&curr);

      // da ora in poi, SIGUSR1 è bloccato di nuovo
      printf("Secondi trascorsi (figlio) %d -- time = %d\n",
             2*seconds, (int)time(NULL));
    }
```

```
60    } else {
        // Stessa cosa per il padre
62      if (seconds == 0)
          sigsuspend(&curr);
64      for (;;) {
          printf("Secondi trascorsi (padre) %d -- time = %d \n",
66              2*seconds-1, (int)time(NULL));
          sleep(1);
68        kill(child, SIGUSR1);
          sigsuspend(&curr);
70      }
    }
72  }
```

In questo caso, anche se il segnale arrivasse tra la `kill()` e la `sigsuspend()`, esso viene *bufferizzato* (ovvero viene segnato come pendente), e nel momento stesso in cui la `sigsuspend()` abilita il segnale ad essere ascoltato, scatta immediatamente l'handler e il segnale viene servito correttamente. Subito dopo essere tornata, la `sigsuspend()` rimette la maschera nello stato in cui era prima della chiamata.

Notare che abbiamo usato la `sigprocmask()` per ottenere lo stato corrente della maschera. Inoltre, la maschera viene ereditata dal figlio.

5.5. Primitiva alarm(), e settaggio dei timeout

Tramite la primitiva `alarm()` è possibile settare un timer che invia un segnale di tipo `SIGALRM` dopo un certo intervallo di tempo.

```
#include <unistd.h>

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

Argomenti in ingresso :

seconds – numero dei secondi dopo i quali scatta il segnale `SIG_ALRM`; se vale 0, allora disattiva il timer.

Valore restituito :

0 – Nel caso in cui il timer non fosse precedentemente settato;

n – Nel caso in cui il timer fosse precedentemente settato, da i secondi rimasti nel settaggio precedente.

La primitiva `alarm()` può essere usata per settare dei timeout su selle operazioni bloccanti. Ad esempio, se stiamo aspettando di leggere un dato da una pipe (vedi il capitolo successivo sulle pipe) ma non vogliamo bloccarci per sempre, allora conviene settare un timeout come segue:

```
int main()
{
  int n, fd;
  char buff[100];

  ...
  alarm(10);
  n = read(fd, buff, 100);

  alarm(0);

  if (n == 0)
    printf("timeout expired!\n");
  else printf("data has been read!\n");

  ...
}
```

5.6. Il segnale SIGCHLD

Il segnale SIGCHLD viene inviato da ogni processo al processo padre al momento della terminazione[1]. In tal modo che il processo padre possa effettuare chiamare la syscall `wait()` per togliere il figlio dallo stato di zombie. Sappiamo come alla morte del padre ogni processo figlio venga ereditato dal processo di sistema `init`; tale processo sfrutta proprio l'handler di `SIGCHLD` per effettuare la `wait()` sui processi rimasti"orfani".

Il comportamento di default per `SIGCLD` e `SIGCHLD` è di ignorare i segnali, cosicché un processo non venga interrotto dalla morte di un figlio.

In certi casi però è opportuno essere notificati della terminazione di un figlio. Supponiamo, che il processo parent crei i processi figli dinamicamente, all'occorrenza di un certo evento. Come vedremo nel capitolo 7, un processo server potrebbe creare un processo figlio ogni volta che arriva una richiesta da un processo client. Il processo figlio ha come unico compito quello di servire la richiesta e poi termina. Se il processo padre non effettua la `wait`, però, il processo figlio rimane nello stato *zombie*.

Un modo di evitare la creazione di zombie, è quello di definire espressamente definire un handler per uno di tali segnali nel quale viene chiamata la syscall `wait()`. Un esempio di tale comportamento viene mostrato nella sezione 7.3.

System V implementa inoltre un particolare comportamento per `SIGCLD`, prevedendo che se un processo dichiara esplicitamente di voler ignorare tale segnale, i figli di tale processo non diverranno zombie (è chiaro che il processo non potrà però più chiamare la syscall `wait()`). In pratica, in un sistema compatibile con System V, una riga del tipo:

```
signal(SIGCLD, SIG\_IGN);
```

[1] Tale segnale è chiamato `SIGCLD` secondo la (vecchia) nomenclatura System V o `SIGCHLD` secondo la più recente nomenclatura BSD, adottata anche da Linux.

previene la creazione di zombie.

Un'altra differenza fra il segnale `SIGCLD` di System V e il segnale `SIGCHLD` è che quando `SIGCLD` è ignorato da un processo, un segnale di tale tipo inviato al processo viene perduto, mentre se `SIGCHLD` è inviato a un processo che lo ignora, viene bufferizzato.

5.7. Comunicazione fra processi

Dopo aver visto come vari processi cooperanti possono sincronizzarsi fra loro, vediamo come possono comunicare. Tipicamente, le primitive di Inter Process Comunication (IPC) possono essere catalogate secondo vari criteri: in particolare, seguendo due criteri fra loro ortogonali, si può distinguere fra comunicazione sincrona (primitive bloccanti) o asincrona (primitive non bloccanti) e diretta (il processo destinatario e la sorgente sono specificati direttamente nelle operazioni di IPC) o indiretta (le comunicazioni non avvengono direttamente, attraverso un'entità intermedia, talvolta detta porta o un canale).

Nei sistemi Unix, nati proprio supportando il paradigma di programmazione a scambio di messaggi, la comunicazione avviene attraverso primitive IPC indirette, generalmente asincrone.

5.7.1. Comunicazione tramite pipe

La forma di IPC più semplice che Unix mette a disposizione è quella ottenibile tramite *pipe*. Un pipe è un canale monodirezionale di comunicazione asincrona, accessibile tramite le primitive di I/O standard messe a disposizione da Unix. Ad ogni pipe sono associati due descrittori di file, uno per l'input ed uno per l'output: i dati scritti sul descrittore di input (tramite la syscall `write()`) vengono bufferizzati all'interno del pipe e possono essere letti dal descrittore di uscita (tramite la `read()`).

Poichè le primitive di output (`write()`) applicate ad un pipe sono non bloccanti (la comunicazione è asincrona), un pipe deve contenere un buffer in cui depositare i dati scritti sulla pipe. Tale buffer non è però infinito, ma ha dimensione `PIPE_BUF` (che è una costante definita in `<unistd.h>`): quando tale buffer è pieno (dopo che `PIPE_BUF` byte sono stati scritti sulla pipe e nessuno è stato letto) la `write()` diventa bloccante.

Un pipe può essere creato tramite la syscall `pipe()`, quindi viene acceduto come se fosse una coppia di normali file (uno aperto in sola lettura ed uno aperto in sola scrittura). Il prototipo della syscall `pipe()` è il seguente:

```
#include <unistd.h>

int pipe(int fd[2]);
```

Argomenti in ingresso :

fd – Array di due interi in cui la primitiva ritorna gli identificatori dei due estremi del pipe.

Valore restituito :

0 – In caso di successo

-1 – In caso di errore.

La syscall crea un pipe e alloca due descrittori di file nella tabella dei file aperti dal processo chiamante, ritornando nell'array `fd` i loro identificatori. In particolare,

`fd[0]` contiene il descrittore dell'uscita del pipe, mentre `fd[1]` contiene il descrittore dell'ingresso. Da questo si capisce come solo il processo che crea il pipe ed i suoi figli abbiano i due estremi della pipe fra le proprie risorse private. Per questo motivo, un pipe può essere utilizzato o per svolgere lavoro inutile (far comunicare un processo con se stesso) o per far comunicare un processo con un suo figlio, o per far comunicare processi discendenti dal processo che ha creato il pipe.

Un utilizzo del pipe può allora essere il seguente:

```
int fd[2];

...
res = pipe(fd);
if (res < 0) {
  perror("Pipe error");
  exit(-1);
}

res = fork();
if (res < 0) sys_err("fork error");

if (fork == 0) {
  ...
  read(fd[0], ...);
  ...
  exit();
}
...
write(fd[1], ...);
...
```

Sebbene un pipe possa essere usato da più di un lettore e da più di uno scrittore contemporaneamente (consentendo comunicazioni del tipo molti/molti, molti/uno, uno/molti e uno/uno), generalmente un solo processo vi accede in lettura ed un solo processo vi accede in scrittura (secondo lo schema di comunicazione uno/uno). In una configurazione di questo tipo, il processo scrittore può chiudere il descrittore di uscita del pipe, mentre il processo lettore può chiudere il processo di ingresso, secondo il seguente schema:

```
int fd[2];

...
res = pipe(fd);
if (res < 0) {
  perror("Pipe error");
  exit(-1);
}

res = fork();
if (res < 0) {
```

```
    perror("Fork error");
    exit(-1);
  }
  if (fork == 0) {
    close(fd[1]);
    ...
  }

  close(fd[0]);
  ...
```

Spesso, il pipe è collegato allo standard input o allo standard output del processo utilizzando la syscall dup(), che duplica un descrittore di file (usando il primo descrittore libero):

```
  ...
  close(0);      /* Chiudo lo stdin */
  dup(fd[0]);    /* E duplico l'output della pipe...          */
                 /* ... che viene cosi' collegata allo stdin */
  close(fd[0]); /* Ora chiudo l'output della pipe...         */
  ...
```

Tutto ciò può essere fatto in modo più semplice usando la syscall `dup2()`. È da notare che Unix fornisce la syscall `popen()` che esegue la sequenza `pipe()`/`fork()`/`dup()`/`exec()`, e `pclose()`, che elimina il pipe ed effettua una `wait()` sul processo figlio. Queste due syscall sono utilizzate dalle shell per implementare catene di processi, a loro volta dette pipe (utilizzando la sintassi "`<prog1> | <prog2>`" i due programmi `<prog1>` e `<prog2>` vengono lanciati in due processi in parallelo, collegando l'uscita del primo con l'ingresso del secondo.

Abbiamo detto che un pipe è monodirezionale, e tipicamente viene usato per comunicazioni uno/uno, quindi l'ingresso del pipe è identificato da un solo descrittore, così come pure l'uscita. Vediamo ora invece cosa succede se tutti i descrittori collegati ad un estremo vengono chiusi.

Se un processo effettua una `read()` su un pipe il cui ingresso non è identificato da nessun descrittore e che ha il buffer vuoto, la `read()` ritorna 0 per indicare la fine del file (End Of File, EOF). Se invece cerca di scrivere (tramite una `write()`) su una pipe la cui uscita non è identificata da nessun descrittore, è generato un segnale `SIGPIPE`. Il processo può gestire tale segnale tramite un apposito handler (specificato tramite `sigaction()`) o ignorarlo: in ogni caso la `write()` verrà interrotta dal segnale, e ritornerà un errore `EPIPE`.

5.7.2. Un Esempio

Se gue un esempio che mostra un semplice caso di utilizzo di un pipe per realizzare una comunicazione monodirezionale fra un processo figlio ed il proprio padre.

Il programma utilizza la tecnica precedentemente spiegata:

1. Crea il pipe
2. Forka il processo figlio, che eredita i descrittori del pipe
3. Poiché il figlio deve inviare dati, chiude il descrittore di ingresso
4. Il padre invece deve ricevere, quindi chiude il descrittore di uscita
5. Quando il figlio ha terminato, chiude anche il descrittore in uscita, ed il padre pu‘øricevere una `SIGPIPE`.

Notare che il corpo del padre e del figlio risiedono nello stesso eseguibile, e non viene usata alcuna funzione della famiglia `exec()`.

```
/* Programma di esempio per l'uso delle pipe.
 * In questo caso semplice il figlio continua
 * ad inviare dati al padre. */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define __USE_POSIX
#include <signal.h>
#include <wait.h>
#include "myutils.h"

/* Dimensione del buffer per la stringa ricevuta
 * dal processo padre */
#define BUFFSIZE 80

/* Scommentare la linea seguente per vedere scattare la SIGPIPE */
#define DO_SIGPIPE

/* Forward declaration */
void handler(int signo);

int main(void)
{
  int pipefd[2], i;
  pid_t child;
  struct sigaction nuova, vecchia;
  char scritta[] = "Prova di scrittura nella PIPE";
  char local[BUFFSIZE];

  /* Installo l'handler della SIGPIPE */
  nuova.sa_handler = handler;
  nuova.sa_flags = 0;
  sigemptyset(&nuova.sa_mask);
  if (sigaction(SIGPIPE, &nuova, &vecchia) == -1)
```

```
    sys_err("Errore nell'installazione della SIGPIPE");

  printf("Lunghezza del buffer della FIFO %ld\n",
         sysconf(_PC_PIPE_BUF));

  if (pipe(pipefd) == -1)
     sys_err("Errore nella creazione della pipe");

  if ((child = fork()) < 0)
     sys_err("Errore nella fork");
  else if (child == 0) {
    /* Il processo figlio chiude il descrittore in lettura poiché
     * sulla pipe deve solo scrivere. */
  close(pipefd[0]);

  /* Ciclo for con cui viene effettuato il passaggio della
   * stringa. */
  for (i=0; i<strlen(scritta); i++) {
    if (write(pipefd[1], &scritta[i], sizeof(char)) == -1)
      sys_err("Errore nella write ...");
  }
  /* Come fine stringa invio un byte a 0 */
  if (write(pipefd[1], 0, sizeof(char)) == -1)
     sys_err("Errore nella write");

  printf("Fine della scritta\n"});
  close(pipefd[1]);
} else {
  /* Il processo padre chiude la pipe in scrittura
   * perche' deve solo leggere .... */
  close(pipefd[1]);
  /* La SIGPIPE scatta quando c'e' una pipe aperta in scrittura
   * ma non in lettura .. infatti in questo caso al task viene
   * notificato, con la SIGPIPE appunto, che
   * potrebbe rimanere indefinitamente bloccato sulla pipe. */
#ifdef DO_SIGPIPE
  close(pipefd[0]);
  /* Nel caso di chiusura della PIPE il padre aspetta la
   * fine del task figlio. */
  wait(NULL);
  exit(0);
#endif
  for (i=0; i<BUFFSIZE; i++) {
    if (read(pipefd[0], &local[i], sizeof(char)) == -1)
      sys_err("Errore nella read ...");
    if (local[i] == 0) break;
  }
  local[BUFFSIZE-1] = 0;
```

```
    printf("Scritta ricevuta -- %s \n", local);
  }
  return 0;
}

void handler(int signo)
{
  printf("Ricevuta signal numero %d \n", signo);
  printf("SIGPIPE = %d \n", SIGPIPE);
  exit(0);
}
```

5.7.3. Comunicazione tramite FIFO

Come già accennato, uno dei più grossi problemi che si possono incontrare utilizzando i pipe è che essi possono essere utilizzati solo dal processo che ha creato il pipe o dai suoi discendenti, in quanto i descrittori dei file che costituiscono l'ingresso e l'uscita del pipe sono privati del processo che lo crea e vengono copiati nei processi creati tramite `fork()`.

Per risolvere tale problema, serve un mezzo per permettere ad un processo di identificare un pipe avendone visibilità globale. Una tipica struttura a visibilità globale condivisa da tutti i processi in un sistema Unix è il file system: per questo motivo molte risorse vengono rimappate su file, in modo da renderle accessibili a tutti i processi che ne conoscano il nome. La soluzione più logica è quindi quella di utilizzare il file system per permettere ai processi di accedere ai pipe: tale soluzione è adottata dai *pipe con nome*, che sono visibili come file. In questo caso i vari processi devono quindi esplicitamente aprire la pipe in ingresso o in uscita (come un normale file) collegando un descrittore all'ingresso o all'uscita del pipe con la syscall `open()`.

I pipe con nome, chiamati anche FIFO, permettono di comunicare a processi "non imparentati" fra loro, purché essi si accordino sul nome del FIFO tramite cui comunicare. Per il resto sono del tutto simili ai normali pipe: canali di comunicazione asincroni e monodirezionali accessibili con le normali primitive di I/O di Unix tramite descrittori di file privati del processo. Anche le FIFO hanno un buffer interno di dimensione limitata, come i pipe semplici.

Come per i pipe, una lettura su una FIFO che non è aperta in scrittura ritorna 0 (EOF), mentre una scrittura su una FIFO che non è aperta in lettura genera un segnale `SIGPIPE` (e la `write()` ritorna `EPIPE`).

Una FIFO può essere creata usando la syscall `mkfifo()`, che ha il seguente prototipo:

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

int mkfifo(const char *pathname, mode_t mode);
```

Argomenti in ingresso :

pathname – Nome della FIFO (all'interno del file system)

mode – Specifica i permessi di accesso alla FIFO.

Valore restituito :

0 – In caso di successo

-1 – In caso di errore.

dove `pathname` è una stringa che indica il nome della FIFO (vista come file). L'accessibilità alla FIFO è controllata in base allo User ID (che contraddistingue ogni utente nel sistema) come specificato dal parametro `mode`, che ha lo stesso significato che ha nella syscall `open()` e può essere ottenuto tramite or aritmetico (l'operatore ||) di varie costanti:

- `S_IRWXU` crea una FIFO accessibile in lettura, scrittura ed esecuzione da parte dell'utente che la possiede;
- `S_IRUSR (S_IREAD)` crea una FIFO accessibile in lettura da parte dell'utente che la possiede;
- `S_IWUSR (S_IWRITE)` crea una FIFO accessibile in scrittura da parte dell'utente che la possiede;
- `S_IRWXG` crea una FIFO accessibile in lettura, scrittura ed esecuzione da parte di tutti gli utenti appartenenti al gruppo che la possiede;
- `S_IRGRP` crea una FIFO accessibile in lettura da parte di tutti gli utenti appartenenti al gruppo che la possiede;
- `S_IWGRP` crea una FIFO accessibile in scrittura da parte di tutti gli utenti appartenenti al gruppo che la possiede;
- `S_IRWXO` crea una FIFO accessibile in lettura, scrittura ed esecuzione da parte di tutti gli utenti;
- `S_IROTH` crea una FIFO accessibile in lettura da parte di tutti gli utenti;
- `S_IWOTH` crea una FIFO accessibile in scrittura da parte di tutti gli utenti;

Una volta che la FIFO è stata creata, i vari processi possono accedervi come ad un normale file, aprendola (creando cioè un descrittore di file che la identifica in lettura o in scrittura) con la syscall `open()` e quindi effettuando `write()` o `read()` sul descrittore ritornato dalla `open()`. Quando si cerca di aprire una FIFO, il risultato dipende dal tipo di accesso che si richiede (lettura o scrittura), dallo stato della FIFO (numero di processi che l'hanno aperta in lettura o in scrittura) e dai flag specificati nella `open()`.

In particolare, se si apre in sola lettura (`O_RDONLY`) una FIFO che non è aperta in scrittura e non si specifica il flag `O_NONBLOCK`, la `open()` è bloccante fino a che qualche

altro processo non apre la FIFO in scrittura. Se invece si specifica il flag `O_NONBLOCK` nella `open()`, la syscall ritorna immediatamente. Analogamente, se si tenta di aprire in sola scrittura una FIFO che non è aperta in lettura e non si specifica il flag `O_NONBLOCK` la `open()` si blocca fino a che un processo non apre la FIFO in lettura. Se invece si specifica il flag `O_NONBLOCK` la `open()` fallisce, con errore `ENXIO`.

5.7.4. Le IPC di System V

System V, una delle due grandi varianti di Unix, propone una serie di primitive di IPC che implementano *message queues* (code di messaggi), *shared memory* (memoria condivisa) e *semaphores* (semafori).

Questi meccanismi non sono standard, ma vengono implementati da kernel molto diffusi come per esempio Linux. Data la loro diffusione e il fatto che sono fra i pochi approcci a consentire l'utilizzo della memoria condivisa, introdurremo anche le IPC di System V, sebbene si discostino abbastanza dalla filosofia di Unix. Per quest'ultimo motivo non andremo però ad approfondire la sintassi e la semantica delle varie primitive, ma ne introdurremo solamente l'utilizzo.

System V individua ogni struttura che permette ai processi di comunicare tramite un identificatore unico nel sistema: in generale, due processi possono comunicare a patto di conoscere entrambi l'identificatore globale della struttura di IPC che intendono utilizzare. Tale valore, o ID, è ritornato dalle syscall usate per creare o per acquisire le strutture di IPC. Vedremo poi quali tecniche possono utilizzare i vari processi per "concordare" sull'identificatore della struttura di IPC da usare.

Ma passiamo ora ad analizzare i meccanismi di IPC di system V, cominciando con le code di messaggi. Una message queue è, come dice il nome, una coda di messaggi collegati fra loro ed accessibili tramite un identificatore di coda (queue ID). Per poter utilizzare una message queue, un processo deve utilizzare la primitiva `msgget()` per creare la coda (o collegarsi ad essa se già esiste).

Poichè le code di messaggi non sono accessibili tramite decrittori di file, le primitive di I/O di Unix non possono essere usate su di esse (e per questo le IPC di System V si discostano parecchio dagli standard Unix), quindi servono delle apposite primitive per accedere ad esse. Tali primitive sono `msgctl()` (equivalente ad `ioctl`), `msgsnd()` (equivalente a `write()`) e `msgrcv()` (equivalente a `read()`). Si invita a riferirsi ai manfile per vedere la sintassi e la semantica di tali primitive.

Oltre alla comunicazione a scambio di messaggi, System V implementa anche il paradigma a memoria comune, discostandosi quindi dal classiso modello di programmazione di Unix. Le primitive di IPC di System V utilizzano un'interfaccia omogenea, quindi anche semafori e regioni di memoria comune saranno creabili e controllabili tramite primitive `*get()` e `*ctl()`.

In particolare, un processo può creare un semaforo (o collegarsi ad esso se già esiste) tramite la syscall `semget()`.

Una volta che un processo ha acquisito un semaforo tramite `semget()`, è in grado di operare su di esso (incrementando o decrementando il suo contatore di valore arbitrario) tramite la primitiva `semop()`.

Per quanto riguarda invece la memoria comune, un processo può ottenere l'identificatore di una zona di memoria comune tramite la primitiva `shmget()`, mapparla nel proprio spazio di memoria privato tramite `shmat()` ed iliminare tale mapping tramite `shmdt()`.

Rimane ora da vedere come processi fra loro indipendenti possano concordare su un ID comune da utilizzare per comunicare. Come detto, l'ID è ritornato dalle primitive `*get`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/shm.h>
#include <sys/sem.h>

int *get(key_t key, ..., int flags);
```

Argomenti in ingresso :

key – Chiave che identifica la struttura di IPC

secondo parametro – `shmget` e `semget` hanno anche un secondo parametro che indica, rispettivamente, la dimensione della zona di memoria condivisa (da arrotondarsi a `PAGE_SIZE`), o il numero di semafori da creare

flags – Flag che specifica i permessi di accesso alla FIFO, ed in più indicano se creare la struttura o collegarsi ad una esistente.

Valore restituito :

ID della struttura di IPC – In caso di successo

-1 – In caso di errore.

Il modo più semplice per fare questo è far generare l'ID dal sistema quando uno dei processi genera la struttura di IPC, e farlo salvare in un luogo noto a tutti i processi che devono comunicare tramite tale struttura di IPC (tipicamente nel file system). Gli altri processi che vogliono accedere a tale struttura dovranno quindi andare a reperire l'ID in tale locazione convenzionale. Un esempio di tale soluzione è quindi il seguente:

```
id = semget(IPC_PRIVATE, 1, 0);

f = fopen("/tmp/miID", "w");
fwrite(f, id);
fclose(f);
...

f = fopen("/tmp/myID", "r");
fread(f, &tmp);
fclose(f);
```

```
id = semget(tmp, 1, 0);
...
```

Un'altra soluzione può essere quella di embeddare una chiave (*key*) che identifichi univocamente la struttura di IPC nel codice dei vari processi: tipicamente un header file incluso da essi conterrà la definizione della chiave come costante. A questo punto la funzione `*get()` convertira' la chiave in un ID. Il problema con questo approccio è che vari set fra loro indipendenti di processi cooperanti possono tentare di usare la stessa chiave per compiti diversi. Questo è inevitabile, ma bisogna comunque evitare che si generino situazioni di inconsistenza: in pratica, è auspicabile che la seconda applicazione che cerca di utilizzare la stessa chiave si blocchi, senza interferire con la prima che ha utilizzato tale chiave. Questo è possibile procedendo nel seguente modo: un solo processo per applicazione crea la struttura di IPC, specificando i flag `IPC_CREAT | IPC_EXCL` che fanno generare una condizione di errore se la struttura già esiste.

La terza soluzione è invece basata sull'utilizzo della funzione `ftok()`, che permette di ricostruire l'ID a partire da una chiave conosciuta dai vari processi (composta da un pathname e da un carattere, il project ID).

Notare che se invece i processi che devono comunicare fra loro sono legati da una relazione padre/figlio, il probelma non sussiste perchè la struttura può essere creata dal padre, e l'ID viene ereditato dal figlio.

5.7.5. Un esempio

Come esempio di utilizzo della memoria condivisa secondo le primitive di IPC di System V portiamo un programma leggermente più complesso del solito. Il programma è composto da due task (padre e figlio) che giocano a tris, utilizzando la memoria condivisa per comunicare.

L'aspetto del programma che ci interessa è appunto la comunicazione attraverso memoria condivisa: in particolare, notare come per condividere l'ID della zona di memoria condivisa i due processi utilizzino la prima delle tre tecniche mostrate sopra (utilizzo di `IPC_PRIVATE` al posto della chiave), avvantaggiandosi della loro relazione padre/figlio (il padre crea la zona di memoria condivisa prima di fare la `fork()`, cosicché il figlio eredita l'ID).

```
1  /* Programma di esempio di memoria condivisa ...
    * In questo caso si tratta di due task che "giocano"
3   * a tris. La strategia e' abbastanza a "caso", nel senso
    * che ogni task mette il suo simbolo a caso in una delle
5   * caselle libere.
    * Dovrebbe risultare abbastanza semplice estenderlo in modo
7   * da avere delle modalita' di gioco piu' complesse.
    * La sincronizzazione in questo caso viene ottenuta per
9   * mezzo di un polling.
    */
11
   #include <stdio.h>
```

```
#include <unistd.h>
#define __USE_POSIX
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <semaphore.h>
#include <fcntl.h>
#include <errno.h>
#include <stdlib.h>
#include <time.h>
#include "myutils.h"


#define CAMPO  "./campo"        /* Nome del file usato per la condivisione */
#define LIBERO '+'       /* indica che la corispondente casella e' libera */
#define PADRE  0             /* indicano chi ha il turno per  giocare ... */
#define FIGLIO 1

/* Defizione e istanziazione di una struttura usata
 * dai due task per giocare. */
struct tris {
  char campo[9];
  int libere;          /* Numero di caselle libere */
  int turno;
} tris;

/* Funzione che serve per stampare a video la schermata
 * con l'attuale situazione del gioco. */
void stampaCampo(char *campo) {
  int i;

  printf("------");
  for (i=0; i<9; i++) {
    if ((i%3) == 0)
      printf("\n|");
    printf("%c|", campo[i]);
  }
  printf("\n------\n");
}

int main(void)
{
  int memd, i, mossa;
  pid_t child;
  struct tris *mioCampo;

  /* Ottengo la zona di memoria condivsa */
```

```
  memd = shmget(IPC_PRIVATE, sizeof(struct tris), SHM_R | SHM_W |
63                IPC_CREAT | IPC_EXCL);

65  if (memd == -1) sys_err("Non sono riuscito a creare la memoria condivisa\n");

67  /* "attacco" la zona di memoria condivisa al mio spazio
     * di indirizzamento */
69  mioCampo = shmat(memd, 0, SHM_R | SHM_W);
    if (mioCampo == -1)
71    sys_err("Non sono riuscito ad agganciare la memoria condivisa\n");

73  /* Inizializzo la zona di memoria condivisa */
    for (i=0;i<9;i++)
75    mioCampo->campo[i] = LIBERO;

77  mioCampo->libere = 9;                    /* Ci sono 9 caselle libere... */
    mioCampo ->turno = PADRE;       /* Il padre inizia a giocare per primo */
79
    if ((child = fork()) < 0) sys_err("Errore nella fork");
81  else if (child == 0) {
      /* Processo figlio .... */
83    while(mioCampo->libere > 0) {
        /* Aspetto in polling che il turno passi al figlio */
85      while(mioCampo->turno == PADRE);

87      /* Estraggo a caso una casella libera */
        mossa = (rand()%mioCampo->libere)+1;
89
        /* Riempio la casella estratta prima con il simbolo "giusto". */
91      for (i=0; i<9; i++) {
          if (mioCampo->campo[i] == LIBERO) {
93          mossa--;
            if (mossa == 0) {
95            mioCampo->campo[i] = 'o';
              break;
97          }
          }
99      }

101     system("clear");                              /* "Pulisco" lo schermo */
        stampaCampo(mioCampo->campo);   /* Stampo a video il "campo" di tris */
103     mioCampo->libere--;       /* Aggiorno il valore delle caselle libere */
        mioCampo->turno = PADRE;             /* Cedo il turno all'altro task */
105     sleep(1);                             /* Fermo il task per 1 secondo */

107   }
      mioCampo->turno = PADRE;  /* In ogni caso prima di uscire cambio turno */
109
      return 0;
```

```
111   }
      else {
113     /* Processo Padre .... */
        while(mioCampo->libere > 0) {
115       /* Aspetto in polling che il turno passi al padre */
          while(mioCampo->turno == FIGLIO);
117
          /* Estraggo caso una delle caselle libere */
119       mossa = (rand()%mioCampo->libere)+1;

121       /* Riempio la casella estratta prima con il simbolo "giusto". */
          for (i=0; i<9; i++) {
123         if (mioCampo->campo[i] == LIBERO) {
              mossa--;
125           if (mossa == 0) {
                mioCampo->campo[i] = 'x';
127             break;
              }
129         }
          }
131
          system("clear");                              /* "Pulisco" lo schermo */
133       stampaCampo(mioCampo->campo);   /* Stampo a video il "campo" di tris */
          mioCampo->libere--;       /* Aggiorno il valore delle caselle libere */
135       mioCampo->turno = FIGLIO;            /* Cedo il turno all'altro task */
          sleep(1);                             /* Fermo il task per 1 secondo */
137     }

139     mioCampo->turno = FIGLIO; /* In ogni caso prima di uscire cambio turno */
        return 0;
141   }
      return 0;
143 }
```

5.8. Esercizi

5.8.1. Segnali

Esercizio 4 – Scrivere un programma che ogni 5 secondi stampa a video un messaggio.

Esercizio 5 – Scrivere un programma che realizza un ciclo infinito in cui non fa niente. Se l'utente preme ctrl-c, il programma stampa a video il numero di secondi trascorsi dall'inizio, e esce.

Esercizio 6 – Ripetere l'esempio "oscillatore" di pagina **??**, cercando di evidenziare i problemi di concorrenza: cercate di farlo andare in deadlock!

In seconda istanza, cercate di ridurre il problema del deadlock (è possibile solo con i segnali eliminare il problema del deadlock?).

Esercizio 7 – Scrivere un programma che realizzi un semplice gioco. Il programma seleziona un numero casuale tra 0 e 100, e l'utente deve indovinare questo numero. Quindi viene realizzato un ciclo, in cui il programma legge da tastiera un numero inserito dall'utente: se il numero è stato indovinato, il gioco finisce; se il numero è maggiore o minore di quello estratto casualmente, viene stampato a video la scritta "maggiore" o "minore", rispettivamente. Se il giocatore non indovina entro 10 secondi, il programma stampa a video "tempo scaduto", e esce.

5.8.2. Pipe

Esercizio 8 – Scrivere un programma in cui, dopo aver aperto due pipe, un processo faccia la fork di un figlio. Da questo momento in poi, ognuno dei due processi, stampa un messaggio a video, manda un dato su una pipe e si mette in atesa sull'altra. Se il programma funziona correttamente, i messaggi del padre e del figlio si devono altrnare sul video.

Esercizio 9 – Scrivere un programma con N processi concorrenti e $N-1$ pipe, fatto nel seguente modo. Il processo padre apre un pipe, forka un figlio e poi si mette in attesa di leggere un intero sulla pipe; quando lo ha ricevuto, lo stampa a video. Ognuno dei figli, tranne l'ultimo, effettua la stessa procedura: apre una pipe, forka un figlio, e si mette in attesa di leggere un intero dalla pipe; non appena riceve l'intero dalla pipe, lo riscrive sulla pipe del padre. L'ultimo figlio, l'N-esimo, legge il proprio pid e lo scrive sulla pipe del proprio padre.

Esercizio 10 – Scrivere un programma che realizzi una struttura circolare con N processi e N pipe. Numerando i processi da 1 a N, il processo i-esimo ($1 < i \leq N$) legge dalla pipe $i-1$ e scrive sulla pipe i. Il processo $i = 1$ legge dalla pipe N e scrive sulla pipe 1. Ognuno dei processi legge e scrive degli interi dalle pipe: i processi dispari leggono un intero, lo stampano a video, gli sommano il proprio indice i e lo scrivono sulla pipe di uscita. I processi pari leggono un intero, lo stampano a video, gli sottraggono il proprio indice e lo inviano sulla pipe di uscita.

Esercizio 11 – Scrivere un programma che stampi a video il contenuto della directory corrente, ottenuto eseguendo il comando di shell ls.

Esercizio 12 – Scrivere un programma che legga due numeri interi dallo stdin e ne stampi a video la somma.

Scrivere un altro programma che manda in esecuzione il programma precedente, passandogli due numeri sullo standard input (utilizzare la primitiva `dup2`).

5.8.3. Pipe con nome

Esercizio 13 – Ripetere l'esercizio 11, con la differenza che per avere il contenuto della directory bisogna inviare una richiesta a un server, che sta in ascolto su una FIFO. In particolare il server deve:

- creare una FIFO di nome "dirFIFO", su cui accetterà le richieste; ongi richiesta consisterà di una stringa contenente il nome della FIFO su cui sarà inviata la risposta.
- ripetere le azioni di cui all'esercizio 11.
- inviare i dati sulla FIFO di risposta.

5.8.4. Vari

Esercizio 14 – Riscrivere il programma dell'esercizio 8, in modo che il processo padre scriva sulla pipe ogni 2 secondi. Come risultato, dovreste vedere 2 messaggi a video, uno del padre e uno del figlio, ogni 2 secondi.

Esercizio 15 – Scrivere un programma concorrente, in cui un processo P0 genera 3 progessi figli, P1, P2 e P3.

Ogni secondo, il processo P0 manda un intero scelto a caso tra 0 e 9 al processo P1 tramite una pipe. Il processo P1 legge il dato, e se è dispari lo manda al processo P2 tramite un'altra pipe, altrimenti lo manda al processo P3 tramite una terza pipe; quindi si mette in attesa di un altro dato da P0.

I processi P2 e P3 si comportano in maniera identica: si mettono in attesa di un dato sulla propria pipe; se ricevono il dato entro 4 secondi, lo stampano a video e si rimettono in attesa da capo; se non ricevono il dato entro 4 secondi, stampano un messaggio a video e terminano.

Dal momento in cui uno tra P2 e P3 termina, anche P0 e P1 devono terminare.

6. Multithreading

In questo capitolo presenteremo un sottoinsieme delle funzioni dello standard POSIX per i thread. Come già detto, POSIX è uno standard della IEEE che specifica l'interfaccia di un sistema operativo. POSIX è uno standard molto complesso, poiché l'interfaccia di un sistema operativo è molto ampia e riguarda vari serivizi (file, IPC, scheduling, networking, etc.). Per questo motivo, lo standard è diviso in sezioni, ognuna della quali si occupa di un sottoinsieme omogeneo di funzioni di interfaccia. Non tutte le sezioni sono obbligatorie: un sistema operativo che possa definirsi POSIX standard può implementare solo alcune delle sezioni dello standard. Solo molto recentemente sistemi operativi POSIX standard implementano la sezione riguardante i thread POSIX. Per esempio, il sistema operativo Linux implementa correttamente la sezione riguardante i thread solo dalla versione 2.6.

Secondo la definizione dello standard POSIX, un processo può contenere uno o più *thread* di esecuzione. Un processo è identificato con uno spazio di indirizzamento privato e da un insieme di proprietà (process ID, file descriptors, etc.). Un thread è un flusso di esecuzione non legato ad uno spazio di memoria privato. Naturalmente un thread ha bisogno di uno spazio di memoria in cui eseguire: è l'intero spazio di memoria assegnato al processo a cui il thread appartiene. In questo senso si può dire che un processo contenente un unico thread sarà un classico processo Unix, come quelli descritti nel Capitolo 2, mentre un processo composto da più thread (che possono quindi comunicare tra loro tramite memoria comune) sarà un processo *multithreaded*.

Dato che tutti i thread creati da un processo girano nello stesso spazio di indirizzamento (quello del processo che li ha creati), essi possono scambiarsi dei dati accedendo alle stesse variabili in memoria, ma anche interferire l'uno con l'altro nel caso in cui la sincronizzazione non sia realizzata correttamente. Naturalmente la gestione della concorrenza fra thread resta onere dell'applicazione.

6.1. POSIX threads

Informalmente parlando, un thread non è altro che una funzione (nel senso che tale termine ha nel linguaggio C) che viene eseguita in maniera concorrente ad altre funzioni, nell'ambito di un processo.

Tutti i thread creati nell'ambito di un processo ne condividono lo spazio di indirizzamento. In aggiunta a questo ogni thread *eredita* dal processo che lo crea i seguenti dati:

- Descrittori dei file,

- Handler dei Signal,
- Directory Corrente,
- ID di utente e di gruppo.

Ogni thread possiede però un suo:

- ID del thread,
- contesto del processore (stack pointer, registri generali, program counter, registro di controllo, ecc.),
- stack,
- maschera dei segnali,
- priorità.

Nei prossimi paragrafi introdurremo con gradualità sia alcuni dei nuovi tipi di dato che alcune delle funzioni implicate nella gestione dei thread dello standard POSIX.

6.1.1. Compilazione di un programma multithreaded

Le funzioni dello standard POSIX vengono di solito fornite tramite una libreria, chiamata libreria `pthread`. I prototipi sono dichiarati negli header file `pthread.h`, `sched.h`, `semaphore.h`. Raccomandiamo sempre di riferirsi al *man pages* tramite il comando man per ottenere informazioni dettagliate sulle primitive.

In alcuni sistemi operativi (ad esempio in alcune versioni di Linux) è necessario specificare esplicitamente la necessità di collegare la libreria dei pthread tramite l'opzione `-lpthread`. Per esempio, per compilare e linkare il programma `myfirstthread.c`, bisogna eseguire il comando:

```
gcc -lpthread myfirstthread.c -o myfirstthread
```

che produce il file eseguibile `myfirstthread`.

6.1.2. Creazione dei Thread

Per supportare il meccanismo dei thread lo standard Posix definisce innanzitutto un nuovo tipo di dato che è il `pthread_t` . Tale nuovo tipo di dato serve per contenere l'identificatore che Unix assegna in modo univoco ad ogni thread.

Ogni processo, prima di iniziare la fase di creazione di uno o più thread, può comunicare al nucleo il numero di thread da creare. Tale numero viene detto *concurrency level*. Ciò può essere fatto invocando la primitiva:

```
#include <pthread.h>

int pthread_setconcurrency (int nThread);
```

Argomenti in ingresso :

nThread – Numero massimo dei thread che si vogliono creare.

Valore restituito :

0 – In caso di successo.

$\neq$ **0** – In caso di insuccesso.

Tale funzione non è affatto obbligatoria, e infatti nella maggior parte delle applicazioni può essere ignorata. In alcuni sistemi embedded con limitate disponibilità di memoria, invece, è necessaria per ottimizzare le strutture dati del sistema operativo e precalcolare la quantità di risorse necessarie al processo.

Un thread è creato invocando la seguente primitiva:

```
#include <pthread.h>

int pthread_create (pthread_t *tid, const pthread_attr_t *attr,
                    (void *(*func)(void *), void *arg);
```

Argomenti in ingresso:

tid Al ritorno dalla chiamata, contiene l'ID del thread

attr Valori passati dell'utente per gli attributi del thread. Nel caso si volessero utilizzare quelli di default si deve passare il valore **NULL**.

func È la funzione che contiene il codice del thread.

arg Parametro opzionale da passare al thread. È un puntatore alla zona di memoria che contiene gli argomenti da passare al thread appena creato.

Valore di ritorno:

0 In caso di successo.

$\neq$ **0** In caso di insuccesso.

Un volta creato un thread può esserci la necessità di sincronizzarsi con il suo istante di terminazione, cioè sospendersi finché il thread non è terminato. Questo è fatto tramite la primitiva:

```
#include <pthread.h>

int pthread_join (pthread_t tid, void **status);
```

Argomenti in ingresso:

tid Identificatore del thread sul cui evento di fine ci si vuole sincronizzare.

status Puntatore all'area di memoria in cui vengono salvati i dati restituiti dal thread.

Valore di ritorno:

0 In caso di successo.

$\neq$ **0** In caso di insuccesso.

Tale funzione ritorna al chiamante solo una volta che il thread il cui ID è passato nel parametro *tid* ha terminato la sua esecuzione. Il nucleo conserva le informazioni su un thread che ha terminato la sua esecuzione fino a quando su quest'ultimo non viene eseguita una `pthread_join`. Questo serve, per esempio, per evitare che un thread si blocchi indefinitamente aspettando un altro thread che ha già terminato la sua esecuzione. D'altra parte però, se non siamo interessati ad eseguire la `pthread_join`, allora si potrebbe aver il fenomeno degli *zombie* (problema del tutto analogo a quello che si verifica nei processi). Per evitare gli zombie, è possibile invocare la primitiva seguente:

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Argomenti in ingresso:

tid id del thread che si vuole sganciare

Valore di ritorno:

0 se la funzione va a buon fine

>0 se c'è un errore (per esempio tid non corrisponde a un thread attivo).

Di solito, la funzione viene chiamata dallo stesso thread che si vuole sganciare con la seguente istruzione:

```
pthread_detach(pthread_self());
```

dove naturalmente la funzione `pthread_self()` restituisce il thread id del thread che invoca la funzione.

La funzione `pthread_kill`, serve per mandare un segnale software a un thread, ed è la corrispondente della `kill()`.

```
#include <pthread.h>

int pthread_kill (pthread_t  tid, int signo);
```

Argomenti in ingresso:

tid – Identificatore del thread a cui si vuole inviare il segnale software.

signo – Identificatore del segnale che si vuole inviare.

Valore di ritorno:

0 – In caso di successo.

$>$ **0** – In caso di insuccesso.

6.1.3. Esempio di creazione di due thread

In questo esempio vedremo come creare due semplici thread. Tutto quello che faranno questi due thread sarà leggere e incrementare una variabile condivisa e quindi stamparla a video.

Dato che un thread altro non è che una funzione C che gira in maniera concorrente ad altre funzioni (nell'esempio i due thread hanno lo stesso codice ed eseguono entrambi la funzione `somma`)), allora si applicano le regole di scope del C standard. Sono quindi condivise fra tutti i thread quelle variabili che sono globali a tutte le funzioni.

Nell'esempio i due thread denominati `th1` e `th2` accedono entrambi alla variabile condivisa `val` incrementandola, entrano quindi in un ciclo pensato solo per far passare del tempo e quindi rileggono e stampano a video la variabile condivisa. Tale accesso non viene regolamentato da alcun meccanismo di gestione delle concorrenza, non si può avere quindi alcuna garanzia sul fatto che il valore che viene alla fine stampato a video sia effettivamente quello che la variabile aveva assunto dopo il suo incremento, poiché, nel frattempo, può essere stato schedulato l'altro thread che l'ha a sua volta modificato.

Oltre a dare un'esemplificazione di come possono essere creati dei thread l'esempio mostra come la concorrenza non regolamentata genera, anche in un caso semplice, problemi di consistenza dei dati, quando a questi si accede senza il supporto di appositi meccanismi

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include "myutil.h"

#define CICLI 10
#define ATTESA_ATTIVA(x,y) for (x=0; x<y;x++)

int val;

/* corpo dei due thread */
void *somma(void *in)
{
  int id = (int) in;
  int i, j;

  for (i=0; i<CICLI; i++) {
    val++;
    ATTESA_ATTIVA(j, 200000);
    printf("Thread %d : somma = %d\n", id, val);
  }
  return 0;
}

int main()
```

```
{
  pthread_t th1, th2;
  int i;
  int r;

  val = 0;

  i = 1;
  r = pthread_create(&th1, 0, somma, (void *)i);
  if (r != 0)
    sys_err("Errore nella creazione del primo thread\n");

  i++;
  r = pthread_create(&th2, 0, somma, (void *)i);
  if (r != 0)
    sys_err("Errore nella creazione del secondo thread\n");

  pthread_join(th1, 0);
  pthread_join(th2, 0);
}
```

6.1.4. Attributi di creazione di un thread

Quando si crea un thread è possibile specificare una serie di *attributi* di creazione, quali la politica di scheduling, la priorità, la dimensione dello stack, etc. Per far questo bisogna innanzitutto dichiarare e inizializzare una variabile di tipo `pthread_attr_t`:

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

Dopo aver utilizzato tale variabile, bisogna distruggerla con la seguente primitiva:

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
```

Nello standard POSIX, vengono specificate almeno 2 politiche di scheduling standard: la politica `SCHED_FIFO` e la politica `SCHED_RR`. Entrambe sono politiche a *priorità fissa*, (ovvero al thread viene assegnata una priorità e il thread a più alta priorità va in esecuzione), e quindi sono politiche di scheduling *real-time*. La differenza fra le due è che mentre nel primo caso due thread alla stessa priorità vengono serviti in modalità FIFO (ovvero il primo arrivato viene servito per primo; il secondo non può andare in esecuzione fino a quando il primo non venga terminato oppure non venga bloccato esplicitamente). Nel secondo caso, tutti i thread allo stesso livello di priorità vengono gestiti in round robin (ovvero, eseguono per un massimo pari al quanto temporale, dopo di che, cedono il processore al successivo thread con la stessa priorità).

Per settare una politicha di scheduling non real-time, bisogna selezionare politiche di scheduling alternative. Ad esempio, nei sistemi Linux e Free BSD, è possibile specificare la politica `SCHED_OTHER` che rappresenta il classico Multi-Level Feedback Queue.

Per settare la politica di scheduling di un thread, bisogna settare l'attributo corrispondente con la seguente funzione:

```
#include <pthread.h>

int pthread_attr_setschedpolicy(pthread_attr_t *a, int policy);
```

Argomenti in ingresso:

a attributi

policy può valere `SCHED_FIFO`, `SCHED_RR` o `SCHED_OTHER`.

E' importante sottolineare che nei sistemi Unix, per creare un thread con una delle due politiche real-time bisogna avere i privilegi di superuser. Se così non fosse, un normale utente potrebbe bloccare il sistema lanciando un programma ad altissima priorità con politica `SCHED_FIFO`, che esegua un ciclo infinito.

Infine, per settare la priorità (o un altro set di parametri di scheduling), è necessario definirsi una struttura di tipo `struct sched_param`, e settare le priorità nel campo `sched_priority`. Il seguente esempio di codice, mostra la creazione di 3 thread con priorità real-time[1].

```
pthread_t th1, th2, th3;
pthread_attr_t my_attr;
struct sched_param param1, param2, param3;

pthread_attr_init(&my_attr);
pthread_attr_setinheritsched(&my_attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&my_attr, SCHED_FIFO);

param1.sched_priority = 1;
param1.sched_priority = 2;
param1.sched_priority = 3;

pthread_attr_setschedparam(&my_attr, &param1);
pthread_create(&th1, &my_attr, body1, 0);

pthread_attr_setschedparam(&my_attr, &param2);
pthread_create(&th2, &my_attr, body2, 0);

pthread_attr_setschedparam(&my_attr, &param3);
pthread_create(&th3, &my_attr, body3, 0);
```

[1] La chiamata alla funzione `pthread_attr_setinheritsched` alla linea 5 è necessaria in alcune versioni del sistema Linux per un bug nella libreria dei thread. Infatti, secondo lo standard POSIX non sarebbe necessaria.

```
22 pthread_attr_destroy(&my_attr);
```

6.1.5. Passaggio parametri

E' possibile passare dei parametri a un thread all'atto della creazione, passandoli come ultimo argomento della `pthread_create`. Per rendere il più generico possibile tale passaggio dei dati, lo standard prevede il passaggio di un puntatore a void. Quindi, la maniera corretta di passare dei dati è la seguente:

- allocare una zona di memoria che contiene i parametri da passare al thread; tale zona di memoria deve essere *dedicata al thread*;
- passare il puntatore a tale zona di memoria.

Il meccanismo viene spiegato più chiaramente con un esempio:

```
int dato; pthread_t tid;

pthread_create(&tid, 0, body, &dato);
```

e nel thread:

```
void *body(void *arg)
{
  int d = * ((int *)arg);

  ...
}
```

dove si fa prima un casting al tipo (int *) e poi si recupera il contenuto della zona di memoria. Attenzione a quando si creano più thread in sequenza!

Come esercizio, spiegare perché il seguente pezzo di codice è errato.

```
pthread_t tid[10]; int i;

for (i=0; i<10; i++)
  pthread_create(&tid[10], 0, body, &i);
```

6.2. Meccanismi di mutua esclusione

Lo standard POSIX supporta meccanismi di mutua esclusione per mezzo del nuovo tipo di dato `pthread_mutex_t`, che è la struttura dati usata per la gestione della mutua esclusione, e di tutta una serie di primitive di nucleo che vi operano.

Un mutex può essere inizializzato staticamente, assegnando alla variabile il valore `PTHREAD_MUTEX_INITIALIZER`, oppure tramite la seguente primitiva:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *m,
                       const pthread_mutex_attr_t *a);
```

Argomenti in ingresso:

m – Indirizzo del mutex.

a – attributi del mutex. Se tale valore è `0`, i valori di dafault vengono utilizzati.

Valore di ritorno:

0 – In caso di successo.

$\neq$ **0** – In caso di insuccesso.

Poiché solo poche implementazioni supportano dei valori significativi del parametro `pthread_attr_t`, nel seguito ci limiteremo a considerare tale parametro sempre uguale a `0`.

Le primitive messe a disposizione dallo standard POSIX sono principalmente due :

```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t * mutex);
```

Argomenti in ingresso:

mutex Indirizzo del semaforo.

Valore di ritorno:

0 In caso di successo.

$>$ **0** In caso di insuccesso.

Queste due funzioni servono rispettivamente per acquisire e rilasciare un semaforo di mutua esclusione. La funzione `pthread_mutex_lock` ritorna al thread chiamante solamente ad acquisizione avvenuta o allo scattare di una condizione di errore. Di questa funzione esiste anche una versione non bloccante che ritorna immediatamente e che ha la seguente sintassi :

```
#include <pthread.h>

int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

Argomenti in ingresso:

mutex Indirizzo del semaforo.

Valore di ritorno:

0 Se il semaforo è stato acquisito.

EBUSY Se il semaforo era già occupato.

È importante sottolineare che le primitive e il tipo di dato appena visti servono solamente per casi di mutua esclusione, e sono soggette a particolari restrizioni. Per esempio, se un thread esegue la `pthread_mutex_lock()`, è lo stesso thread che *deve* eseguire la `pthread_mutex_unlock()` corrispondente.

Quindi, non è possibile utilizzare la tecnica del *passaggio del testimone* con i `pthread_mutex_t`. Infatti, tale tecnica richiede che un thread che sia dentro una sezione critica e svegli un altro thread bloccato su una condizione, invece di rilasciare la sezione critica, semplicemente "passi il testimone" al thread appena svegliato che si occuperà di rilasciare la sezione critica più tardi. Quindi la tecnica del passaggio del testimone prevede che la sezione critica sia iniziata da un thread e terminata da un altro. Poiché non è legale eseguire la `pthread_mutex_lock()` in un thread e la corrispondente `pthread_mutex_unlock()` in un altro thread, la tecnica del passaggio del testimone non può essere realizzata con i mutex. Per realizzare tale tecnica è necessario utilizzare direttamente il meccanismo semaforico.

Invece, come vedremo meglio più avanti, i `pthread_mutex_t` insieme alle variabili condition, simulano il comportamento del costrutto *monitor*.

Altre note importanti:

- Le primitive sui mutex non sono punti di cancellazione (*cancellation points*). Quindi, se un thread viene terminato mentre tiene un `pthread_mutex_lock()` e prima che abbia rilasciato il mutex, è onere del corrispondente handler di rilasciare il mutex (Vedi più avanti per i cancellation points).

- Le primitive relative ai mutex non possono essere invocate dall'interno di un signal handler, pena un possibile deadlock.

6.2.1. Esempio: meccanismi di mutua esclusione

In questo esempio vedremo il più semplice meccanismo di gestione della concorrenza, che è la mutua esclusione.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include "myutils.h"

#define CICLI 10
#define LUN 50
#define ATTESA_ATTIVA(x,y) for(x=0; x<y; x++)
```

```
12 struct {
     pthread_mutex_t mutex;
14   char scritta[LUN+1];
     int numLetture;
16   int numScritture;
   } shared;
18
   void *scrittore(void *arg)
20 {
     int carattere = (char)arg;
22   int i, j, k;

24   for (i=0; i<CICLI; i++) {
       while(1) {
26       pthread_mutex_lock(&shared.mutex);
         if (shared.numLetture == shared.numScritture) {
28         break;
         }
30       pthread_mutex_unlock(&shared.mutex);
       }
32     for(k=0; k<LUN; k++)
         shared.scritta[k] = carattere;
34     shared.numScritture++;
       pthread_mutex_unlock(&shared.mutex);
36     ATTESA_ATTIVA(j, 200000);
     }
38 }

40 void *lettore(void *arg)
   {
42   int i;

44   for (i=0; i<2*CICLI; i++) {
       while(1) {
46       pthread_mutex_lock(&shared.mutex);
         if (shared.numLetture != shared.numScritture)
48         break;
         pthread_mutex_unlock(&shared.mutex);
50     }
       printf("Lettore -- scritta = %s\n", shared.scritta);
52     shared.numLetture++;
       pthread_mutex_unlock(&shared.mutex);
54   }
   }
56
   int main(void)
58 {
     pthread_t scrit1, scrit2, let;
60   int res, i;
```

```
62    /* Inizializzo la stringa scritta */
      for(i=0; i<LUN; i++)
64      shared.scritta[i] = 'x';

66    /* Ogni stringa C deve terminare con 0!!!!! */
      shared.scritta[LUN] = 0;
68    shared.numLetture = 0;
      shared.numScritture = 0;
70
      /* Inizializzazione mutex */
72    pthread_mutex_init(&shared.mutex, 0);

74    /* A questo punto posso creare i thread .... */
      res = pthread_create(&scrit1, 0, scrittore, (void *)'+');
76    if (res != 0)
        sys_err("Errore nella creazione del primo thread\n");
78
      res = pthread_create(&scrit2, 0, scrittore, (void *)'-');
80    if (res != 0) {
        printf("Errore nella creazione del secondo thread\n");
82      pthread_kill(scrit1, SIGKILL);
        return -1;
84    }

86    res = pthread_create(&let, 0, lettore, 0);
      if (res != 0) {
88      printf("Errore nella creazione del secondo thread\n");
        pthread_kill(scrit1, SIGKILL);
90      pthread_kill(scrit2, SIGKILL);
        return -1;
92    }

94    /* A questo punto aspetto che i due thread finiscano ... */
      pthread_join(scrit1, 0);
96    pthread_join(scrit2, 0);
      pthread_join(let, 0);
98  }
```

In questo caso si hanno due thread (che eseguono lo stesso codice rappresentato dalla funzione `scrittore`) che ciclicamente scrivono una stringa in un buffer condiviso, aspettando un certo tempo fra una scrittura e l'altra, e un thread lettore che si deve sincronizzare con l'istante di fine scrittura sul buffer e quindi stampare la stringa risultante. L'esempio è congegnato in modo che un thread scrittore non può riscrivere i dati del buffer se prima il thread lettore non ne ha fatto al relativa stampa a video.

Nell'esempio non abbiamo ancora introdotto le funzioni per la sincronizzazione, quindi la sincronizzazione sulla condizione di '*fine scrittura stringa*' può essere fatta solamemente tramite un polling.

Esercizio: Si risalga al file include in cui è definita la macro `PTHREAD_MUTEX_INITIALIZER` e si verifichi la sua consistenza con la definizione della struttura `pthread_mutex_t`.

6.3. Attesa di una condizione

Come detto nella sezione 6.2 con i soli meccanismi di mutua esclusione l'unico meccanismo di attesa per il verificarsi di una certa condizione è quello del polling. Tale meccanismo è ovviamente dispendioso dal punto computazionale oltre che poco elegante.

Per ovviare a tale inconveniente lo standard POSIX mette a disposizione un ulteriore meccanismo per consentire a un thread di bloccarsi in attesa che una certa condizione diventi vera.

Tale meccanismo si avvale della definizione di un nuovo tipo di dato (`pthread_cond_t`), contenente i campi necessari alla gestione del bloccaggio/sbloccaggio dei thread. Una variabile di tipo di tipo `pthread_cond_t` si chiama *variabile condizione* ed è associata in maniera implicita ad una condizione logica che si vuole verificare.

Le funzioni che operano su tale struttura dati sono :

```
#include <pthread.h>

int pthread_cond_wait (pthread_cond_t *cond_var,
                       pthread_mutex_t *mutex);
```

Argomenti in ingresso:

cond – Indirizzo di una istanza del tipo `pthread_cond_t` che rappresenta la condizione di sincronizzazione.

mutex – Semaforo di mutua esclusione necessario alla gestione corretta consistenza dei dati.

Valore di ritorno:

0 – In caso di successo

$\neq$ **0** – In caso di insuccesso.

La `pthread_cond_wait` serve per sincronizzarsi con una certa condizione all'interno di un blocco di dati condivisi e protetti da un semaforo di mutua esclusione. La presenza, fra gli input, del semaforo di mutua esclusione, nel caso della wait, serve a garantire che al momento del bloccaggio esso venga liberato per consentire ad altri thread di potere accedere alla zona di memoria condivisa. Inoltre, se la wait ritorna in modo regolare, allora il sistema garantisce che anche il semaforo di mutua esclusione sia stato nuovamente acquisito.

Si tenga inoltre presente che la funzione signal **non** libera il semaforo di mutua esclusione, poiché esso non è tra i suoi argomenti in ingresso, e quindi questo deve essere esplicitamente liberato per consentire l'accesso ai dati, in caso contrario si rischiano situazioni di deadlock.

Quindi, tramite utilizzando congiuntamente le variabili condition e i mutex è possibile simulare il costrutto *monitor* presenti in alcuni linguaggi dedicati alla concorrenza. É bene tenere presente che una variabile condition non è un semaforo. Quindi se nessun thread è bloccato sulla condizione, una `pthread_cond_signal()` su tale condizione non ha alcun effetto.

6.3.1. Un esempio: meccanismi di sincronizzazione su una condizione

In questo esempio vedremo come dei thread possano bloccarsi in attesa che una certa condizione, associata in maniera implicita a una variabile di tipo `pthread_cond_t` sia verificata.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include "myutils.h"

#define CICLI 10
#define LUN 20
#define DELAY_WRITER 200000
#define DELAY_READER 2000000
#define ATTESA_ATTIVA(x,y) for(x=0; x<y; x++)

struct {
  pthread_mutex_t mutex;
  pthread_cond_t lettore;
  pthread_cond_t scrittori;
  char scritta[LUN+1];
  int primo, ultimo, elementi;
  int blockscri, blocklet;
} shared;

/* Forward declaration */
void *scrittore(void *);
void *lettore(void *);

int main(void)
{
  pthread_t s1TID, s2TID, lTID;
  int res, i;

  /* Inizializzo la stringa scritta */
  for(i=0; i<LUN; i++)
    shared.scritta[i] = 'x';
  /* Ogni stringa C deve terminare con 0!!!!! */
  shared.scritta[LUN] = 0;
  shared.primo = shared.ultimo = shared.elementi = 0;
```

```
38    shared.blocklet = shared.blockscri = 0;

40    pthread_mutex_init(&shared.mutex, 0);
      pthread_cond_init(&shared.lettore, 0);
42    pthread_cond_init(&shared.scrittori, 0);

44    /* A questo punto posso creare i thread .... */
      res = pthread_create(&lTID, 0, lettore, (void *)'+');
46    if (res != 0) sys_err("Errore nella creazione del primo thread\n");

48    res = pthread_create(&s1TID, 0, scrittore, (void *)'-');
      if (res != 0) {
50      printf("Errore nella creazione del secondo thread\n");
        pthread_kill(s1TID, SIGKILL);
52      exit(-1);
      }
54
      res = pthread_create(&s2TID, 0, scrittore, 0);
56    if (res != 0) {
        printf("Errore nella creazione del terzo thread\n");
58      pthread_kill(lTID, SIGKILL);
        pthread_kill(s1TID, SIGKILL);
60      return -1;
      }
62
      /* A questo punto aspetto che i tre thread finiscano ... */
64    pthread_join(s1TID, 0);
      pthread_join(s2TID, 0);
66    pthread_join(lTID, 0);
  }
68
  /* Codice relativo a uno dei thread che scrivono */
70 void *scrittore(void *arg)
  {
72    char carattere = (char) arg;
      int i, j, k;
74
      for (i=0; i<CICLI; i++) {
76      for(k=0; k<LUN; k++) {
          pthread_mutex_lock(&shared.mutex);
78        while (shared.elementi == LUN) {
            shared.blockscri++;
80          pthread_cond_wait(&shared.scrittori, &shared.mutex);
            shared.blockscri--;
82        }
          shared.scritta[shared.ultimo] = carattere;
84        shared.ultimo = (shared.ultimo+1)%(LUN);
          shared.elementi++;
86        if (shared.blocklet != 0)
```

```
        pthread_cond_signal(&shared.lettore);
88      pthread_mutex_unlock(&shared.mutex);
        ATTESA_ATTIVA(j, DELAY_WRITER);
90    }
  }
92}

94void *lettore(void *arg)
{
96  int i, k, j;
  char local[LUN+1];
98
  local[LUN] = 0;
100
  for (i=0; i<2*CICLI; i++) {
102   for (k=0; k<LUN; k++) {
      pthread_mutex_lock(&shared.mutex);
104   while (shared.elementi == 0) {
        shared.blocklet++;
106     pthread_cond_wait(&shared.lettore, &shared.mutex);
        shared.blocklet--;
108   }
      local[k] = shared.scritta[shared.primo];
110   shared.primo = (shared.primo+1)%(LUN);
      shared.elementi--;
112
      if (shared.blockscri != 0)
114     pthread_cond_signal(&shared.scrittori);

116   pthread_mutex_unlock(&shared.mutex);
      ATTESA_ATTIVA(j, DELAY_READER);
118
    }
120 printf("Stringa = %s \n", local);
  }
122}
```

In questo caso gli scrittori inseriscono i dati in un buffer circolare di lunghezza definita dalla macro **LUN**, e si bloccano solo se il buffer è già pieno. L'unico thread che legge continua ad estrarre caratteri e si blocca solamente quando il buffer risulta essere vuoto.

Le strutture dati che si sono viste fino ad ora sono più che sufficienti per bloccare un thread in attesa che una certa condizione si verifichi. Appare anche più che logica la scelta di dividere i semafori nelle due tipologie di *semafori di mutua esclusione* e *semafori di condizione*. Manca però ancora un meccanismo sintattico che sia simile alle primitive semaforiche viste durante il corso e che fanno parte dei trattati classici di Sistemi Operativi.

Esercizio: Nell'esempio è necessario che il thread lettore ottenga anche il semaforo di mutua esclusione? La risposta cambierebbe se ci fossero due thread che leggono?

Giustificare la risposta con degli esempi.

Esercizio: come mai le condizioni vengono controllate all'interno di un ciclo `while`? sapreste dare un esempio di funzionamento non corretto nel caso in cui gli scrittori controllassero le condizioni con un semplice `if` invece che con un ciclo `while`?

6.4. Semafori

Per supportare un meccanismo semaforico generico, lo standard POSIX offre un nuovo tipo di dato, `sem_t` che è un'implementazione del classico meccanismo semaforico. Meccanismi analoghi potrebbero essere costruiti di volta in volta sfruttando le primitive e i tipi di dato già visti (è il caso dell'esempio 6.3.1) ma questo complicherebbe ogni volta la scrittura del programma. Si deve considerare questo tipo di dato, e le relative primitive associate, come un'utile scorciatoia offerta al programmatore.

Data la sua diversa natura non è stata prevista una macro di inizializzazione di un semaforo, per cui esso dovrà essere esplicitamente inizializzato con il valore voluto tramite la primitiva:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared,
             unsigned int value);
```

Argomenti in ingresso:

sem – Indirizzo del semaforo che si vuole inizializzare.

pshared – Se posto a zero indica che il semaforo non è condiviso fra più processi, altrimenti indica che il semaforo deve essere condiviso fra più processi[2].

value – Valore iniziale del semaforo.

Valore di ritorno:

0 – In caso di successo

-1 – In caso di insuccesso.

Dopo aver istanziato una variabile di tipo `sem_t` ed averla inizializzata con la primtiva appena vista potremo operare su di essa con le seguenti primitive :

```
#include <semaphore.h>

int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

Argomenti in ingresso:

sem – Indirizzo del semaforo su cui si vuole operare.

[2] Allo stato attuale l'implementazione dei LinuxThreads supporta solamente $pshared = 0$

Valore di ritorno:

0 – In caso di successo

-1 – In caso di insuccesso.

Queste due funzioni rappresentano il modo *classico* di operare su dei semafori per cui non ci si dilungherà ulteriormente sul loro uso.

```
#include <semaphore.h>

int sem_trywait(sem_t * sem);
```

Tale primitiva è la versione non bloccante della `sem_wait`.

Argomenti in ingresso:

sem – Indirizzo del semaforo su cui si vuole fare la wait non bloccante.

Lo standard **Posix** definisce anche una primitiva per potere leggere il valore attuale di un semaforo :

```
#include <semaphore.h>

int sem_getvalue(sem_t *sem, int *sval));
```

Argomenti in ingresso:

sem – Indirizzo del semaforo di cui si vuole leggere il valore.

sval – Puntatore all'intero in cui viene scritto il valore del semaforo.

Per entrambe le primitive vale quanto visto sopra riguardo al valore di ritorno.

6.4.1. Un esempio

In questo esempio vedremo un uso base dell'implementazione dei semafori dello standard POSIX.

```
1  #include <semaphore.h>
   #include <pthread.h>
3  #include <signal.h>
   #include <stdio.h>
5
   #define CICLI 1
7  #define LUN 20
   #define DELAY_WRITER 200000
9  #define DELAY_READER 2000000
   #define ATTESA_ATTIVA(x,y) for(x=0; x<y; x++)
11
   struct {
13   char scritta[LUN+1];
     int primo, ultimo;
```

```
15    sem_t mutex, piene, vuote;
  } shared;
17
  void *scrittore(void *);
19 void *lettore(void *);

21 int main(void)
  {
23    pthread_t s1TID, s2TID, lTID;
      int res, i;
25
      for(i=0; i<LUN; i++)
27      shared.scritta[i] = 'x';

29    shared.primo = shared.ultimo = 0;

31    /* Fase di inizializzazione dei semafori */
      if (sem_init(&shared.mutex, 0, 1) == -1)
33      sys_err("Non sono riuscito ad inizializzare shared.mutex\n");
      if (sem_init(&shared.piene, 0, 0) == -1)
35      sys_err("Non sono riuscito ad inizializzare shared.piene\n");

37    if (sem_init(&shared.vuote, 0, LUN) == -1)
        sys_err("Non sono riuscito ad inizializzare shared.vuote\n");
39
      res = pthread_create(&lTID, 0, lettore, 0);
41    if (res != 0) sys_err("Errore nella creazione del primo thread\n");

43    res = pthread_create(&s1TID, 0, scrittore, (void *)'+');
      if (res != 0) {
45      printf("Errore nella creazione del secondo thread\n");
        pthread_kill(s1TID, SIGKILL);
47      return -1;
      }
49
      res = pthread_create(&s2TID, 0, scrittore, (void *)'-');
51    if (res != 0) {
        printf("Errore nella creazione del terzo thread\n");
53      pthread_kill(lTID, SIGKILL);
        pthread_kill(s1TID, SIGKILL);
55      return -1;
      }
57
      /* A questo punto aspetto che i tre thread finiscano ... */
59    pthread_join(s1TID, 0);
      pthread_join(s2TID, 0);
61    pthread_join(lTID, 0);
  }
63
```

```
void *scrittore(void *arg)
{
  char carattere = (char) arg;
  int i, j, k;

  for (i=0; i<CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.vuote);
      sem_wait(&shared.mutex);
      shared.scritta[shared.ultimo] = carattere;
      shared.ultimo = (shared.ultimo+1)%(LUN);
      sem_post(&shared.mutex);
      sem_post(&shared.piene);
      ATTESA_ATTIVA(j, DELAY_WRITER);
    }
  }
  return 0;
}

void *lettore(void *in)
{
  int i, j, k;
  char local[LUN+1];

  local[LUN] = 0;

  for (i=0; i<2*CICLI; i++) {
    for(k=0; k<LUN; k++) {
      sem_wait(&shared.piene);
      sem_wait(&shared.mutex);
      local[k] = shared.scritta[shared.primo];
      shared.primo = (shared.primo+1)%(LUN);
      sem_post(&shared.mutex);
      sem_post(&shared.vuote);
      ATTESA_ATTIVA(j, DELAY_READER);
    }
    printf("Stringa = %s \n", local);
  }
}
```

L'esempio è essenzialmente analogo all'esempio precedente, la differenza sostanziale è rappresentata dal fatto che il codice dei thread risulta molto più snello poichè molte operazioni sono compiute implicitamente dalle nuove primitive associate al tipo `sem_t`.

6.5. Punti di cancellazione

Un thread può richiedere ad un altro thread di terminare in maniera "pulita", cioè rilasciando le risorse eventualmente possedute e lasciando le strutture dati condivise in

maniera consistente.

Per questo motivo vengono definite una serie di funzioni:

```
#include <pthread.h>

int pthread_cancel(pthread_t t);
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
int pthread_testcancel(void);
```

Più precisamente, un thread può inviare una richiesta di cancellazione au un altro thread tramite la primitiva `pthread_cancel()`. A seconda dei suoi settaggi, il thread può ignorare la richiesta, oppure terminare immediatamente, oppure sospendere la richiesta fino a che non si arriva a un "cancellation point".

Un thread può specificare il suo comportamento tramite la primitiva `pthread_setcancelstate()`. Il primo argomento di tale primitiva può essere `PTHREAD_CANCEL_ENABLE` (che significa che le richieste d icancellazione vengono servite) oppure `PTHREAD_CANCEL_DISABLE` (che significa che le richieste vengono ignorate). Il vecchio stato viene restituito nel secondo argomento. Tramite la `pthread_setcanceltype()`, è possibile specificare se la richiesta deve essere servita immediatamente (`PTHREAD_CANCEL_ASYNCHRONOUS`), oppure ritardata fino al primo punto di cancellazione (`PTHREAD_CANCEL_DEFERRED`).

I punti di cancellazione sono i punti nel codice in cui viene chiamata una delle seguenti funzioni:

```
  pthread_join();
  pthread_cond_wait();
  pthread_testcancel();
  sem_wait();
```

La funzione `pthread_testcancel()` non fa altro che testare se c'è una richiesta di cancellazione pendente.

Una cancellazione è equivalente in tutto e per tutto a una chiamata a una `pthread_exit()`. Per assicurarsi che le strutture dati siano sempre in uno stato consistente, è spesso necessario operare delle operazioni di "pulizia" quando un thread termina. Questo può essere fatto con le seguenti funzioni:

```
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *), void *arg);
void pthread_cleanup_pop(int execute);
```

La prima funzione serve ad installare un handler che viene chiamato non appena il thread termina o per una chiamata a `pthread_exit()` oppure per una cancellazione. Di solito essa viene chiamata più volte in quanto è opportuno installare un handler per ogni risorsa che potrebbe rimanere in uno stato inconsistente. La corrispondente funzione `pthread_cleanup_pop` serve a rimuovere l'ultimo handler installato. Se il suo argomento è diverso da 0, esegue l'handler prima di rimuoverlo.

Gli handler sono della forma:

```
void hand(void *arg);
```

e l'argomento che viene passato è il secondo argomento della `pthread_cleanup_push()`.
NOTA BENE: Le funzioni `pthread_cleanup_push()` e `pthread_cleanup_pop()` sono in realtà delle macro. La `pthread_cleanup_push()` lascia una parentesi { aperta che viene chiusa dalla corrispondente funzione `pthread_cleanup_pop()`. Quindi, esse devono assolutamente essere chiamate nella stesso blocco di codice, cosicché le parentesi abbiano un match. Di solito, la `pthread_cleanup_push()` viene chiamata all'inizio del thread e la `pthread_cleanup_pop()` subito prima della parentesi di chiusura del thread.

Nel seguente esempio, viene mostrato come sia possibile "proteggersi" da una eventuale richiesta di cancellazione che arrivi tra una `pthread_mutex_lock()` e la corrispondente `pthread_mutex_unlock()`.

```
pthread_mutex_t m;
int type, oldtype;
...

pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
pthread_cleanup_push(pthread_mutex_unlock, (void *)&m);
pthread_mutex_lock(&m);
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &type);
// sezione critica
// ...
// ...
pthread_cleanup_pop(1);
```

7. Sistemi distribuiti

In questo capitolo tratteremo il modello "client/server" (cliente servitore) per l'interazione tra processi tramite l'interfaccia dei socket. Una trattazione completa di tali argomenti esula da queste dispense. Al lettore interessato ad approfondire consigliamo l'ottimo testo di Stevens [1].

Diamo per scontato che lo studente conosca la suite di protocolli di comunicazione TCP/IP e UDP/IP.

7.1. Modello client/server

Nel modello client server, due entità che intendono comunicare, siano A e B, hanno ruoli diversi. Una viene chiamata "server" (ad esempio A) e una viene chiamata client (ad esempio B). Il client B è quello che inizia la comunicazione inviando una richiesta al server. Il processo server A è inizialmente bloccato in attesa di richieste di connessione. L'avvio delle comunicazione è quindi asimmetrico: il client ha bisogno di conoscere l'indirizzo del server e il protocollo di comunicazione con cui intende comunicare, mentre il server sta semplicemente in attesa di richieste da un qualsivoglia cliente. Una buona analogia è quella di una persona che fa una chiamata telefonica ad una altra persona, e di cui deve necessariamente conoscere il numero di telefono.

Una volta che il canale di comunicazione viene stabilito, sia A che B possono inviare e ricevere informazioni.

7.2. L'interfaccia dei socket

Un socket è un canale di comunicazione tra processi, e quindi simile come concetto di base ai meccanismi di IPC come le FIFO viste nel capitolo 5.7. Un socket è però molto più complesso e flessibile. Infatti è possibile specificare il protocollo di comunicazione; inoltre è possibile stabilire comunicazioni tra processi che risiedono su nodi computazionali diversi. Dal punto di vista del suo uso, a seconda del tipo di protocollo, si usano primitive diverse per stabilire una canale di comunicazione, e per mandare e ricevere messaggi.

In queste dispense ci limiteremo a descrivere l'interfaccia dei socket per i protocolli di Internet, e tralasceremo i socket di dominio Unix.

Come tipo di dati, un socket è un descrittore di file, che può essere creato con la seguente primitiva:

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

```
int socket(int family, int type, int protocol);
```

Argomenti in ingresso:

family Specifica la famiglia di protocolli, e può valere una delle costanti riportate nella Tabella 7.1.

type è una delle costanti definite in tabella 7.2 e specifica il tipo di connessione da realizzare.

protocol è il protocollo che si vuole utilizzare, e di solito questo parametro si mette pari a 0 (che indica il protocollo di default per la coppia famiglia/tipo specificata dai due paramteri precedenti).

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	Key sockets

Tabella 7.1.: Famiglie di protocolli

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Tabella 7.2.: Type del socket per la funzione `socket`

In queste dispense, faremo sempre uso della combinazione `family=AF_INET` e `type=SOCK_STREAM` che corrisponde a una comunicazione con il protocollo internet TCP/IP. Più avanti vedremo molto sinteticamente la combinazione `family=AF_INET` e `type=SOCK_DGRAM` che corrisponde al protocollo internet UDP/IP.

Se la chiamata alla funzione `socket` va a buon fine, viene restituito un *file descriptor* che può essere usato per comunicare tra client e server. Sia il client che il server devono effettuare le rispettive chiamate a tale funzione. Comunque, una semplice chiamata alla funzione `socket` non è ancora sufficiente a stabilire una connessione. Il client e il server a questo punto devono fare degli ulteriori passi prima di poter scambiare dei dati.

7.2.1. Socket TCP/IP

Poiché il processo di creazione di una connessione è piuttosto differente da client a server, in Figura 7.1 riportiamo lo schema principale di chiamate di primitive che il client e il server devono effettuare per effettuare una connessione tramite il protocollo TCP/IP.

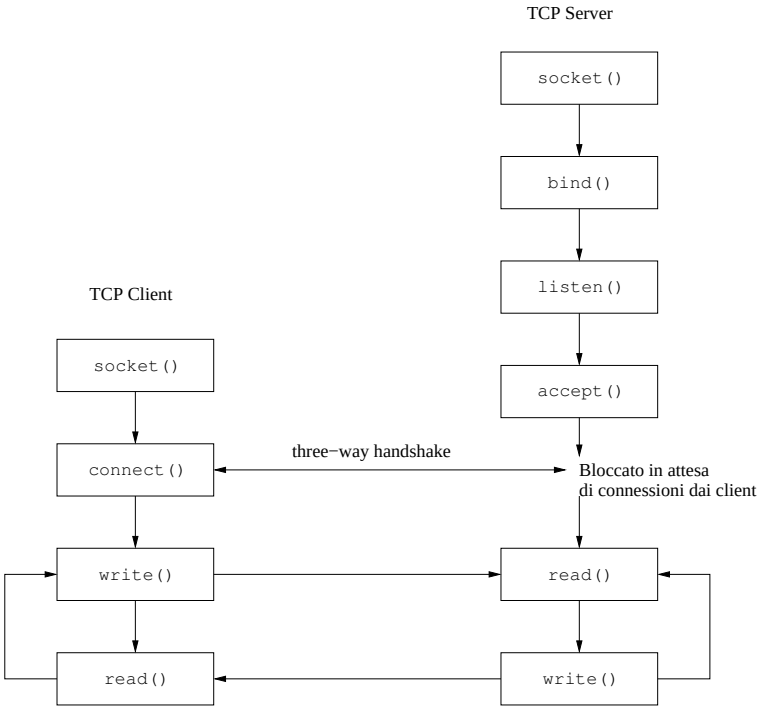

Figura 7.1.: Passi necessari a stabilire una connessione TCP/IP

Cominciamo a studiare il lato server.

Funzione bind Il server deve innanzitutto effettuare una chiamata alla funzione `bind()`.

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

int bind(int sockfd, const struct sockaddr *my, socklen_t addrlen);
```

Argomenti in ingresso:

sockfd descrittore del socket (valore di ritorno della funzione `socket()`

my indirizzo del server, comprendente un indirizzo IP e una porta di connessione

addrlen dimenzione in byte della struttura di cui al parametro precedente.

Un indirizzo internet è composto di un indirizzo IP (composta a sua volta di 4 byte) e una porta (un intero senza segno a 16 bit, quindi da 0 a 65535). Gli indirizzi internet vengono anche chiamati *end-point*. Un esempio di end-point è **196.205.82.15/1500**, dove la prima parte è l'indirizzo IP, e la seconda parte rappresenta la porta di connessione.

Le porte da 0 a 1023 sono porte riservate ad applicazioni ben specifiche (*well-known ports*). La lista delle porte assegnate e le relative applicazioni è riportata nel RFC[1]

[1] *Request for Comments*

1700, e continuamente aggiornata dallo IANA (*Internet Assigned Numbers Authority*), un'associazione internazionale che si occupa di assegnare gli indirizzi numerici di Internet.

Per esempio, il server SSH è di solito in ascolto sulla porta 22; il server http è in ascolto sulla porta 80; e così via. Nulla vieta però di spostare server ben conosciuti su porte non convenzionali (ad esempio spostare il server telnet su una porta "alta").

Le porte da 1024 a 49151 sono dette *registered ports* liberamente utilizzabili da applicazioni ben definite. Lo IANA tiene traccia delle convenzioni utilizzate per queste porte e riporta delle liste provvisiorie di associazioni porte/applicazioni. Lo IANA però non ha alcun potere su questo intervallo di porte. Ad esempio, alcuni applicativi web, come Zope, utilizzano la porta 8080, che è ormai diventata uno standard per questo tipo di servizi.

Infine, le porte da 49151 a 65535 sono assegnate dal sistema operativo ai processi client, e vengono denominate *ephemeral ports*, in quanto la loro associazione con un programma è dinamica.

Per specificare l'indirizzo nella bind, bisogna riempire una struttura di tipo:

```
struct sockaddr_in {
  uint8_t         sin_len;      /* lun. della strutt. */
  sa_family       sin_family;   /* AF_INET */
  in_port_t       sin_port;     /* porta  (16 bit) */
  struct in_addr  sin_addr;     /* indirizzo IPv4 */
  char            sin_zero[8];  /* non usato */
};
```

A sua volta, il campo `sin_addr` è definito come:

```
struct in_addr {
  in_addr_t   s_addr; /* indirizzo IPv4 (32 bit) */
};
```

Tali strutture sono definite nell'header file `<netinet/in.h>`. Di solito, l'indirizzo IP del nodo su cui il server esegue è unico, e quindi specificarlo esplicitamente è inutile, e inoltre renderebbe il programma poco portabile: se si cambia l'indirizzo del nodo, o se si fa girare il programma server su tanti nodi diversi bisognerebbe specificare ogni volta l'indirizzo IP del nodo in questione. Infine, è possibile che a un nodo server corrispondano più indirizzi IP diversi. In questi casi, è possibile che il programma server voglia accettare richieste provenienti da tutti gli indirizzi IP assegnati al nodo.

Per semplificare la procedura di assegnamento, è possibile impostare il campo `s_addr` al valore di default `INADDR_ANY`, che corrisponde a una sorta di *wild card*, ovvero si dice che da qualunque indirizzo IP provenga la richiesta, essa può essere accettata. In un server che esegue su un nodo con un unico indirizzo IP, questo equivale a dire che si accettano connessioni sull'indirizzo IP del nodo; se il nodo ha più indirizzi IP, equivale a dire che si accettano connessioni da *tutti* gli indirizzi IP del nodo.

Un tipico modo di specificare l'indirizzo per un server è il seguente:

```
struct sockaddr_in my_addr;
int myport = 50000;
```

```
4   bzero(&my_addr, sizeof(struct sockaddr_in));
    my_addr.sin_family = AF_INET;
6   my_addr.sin_port = htons(myport);
    my_addr.sin_addr.s_addr = INADDR_ANY;
8   base_sd = socket(AF_INET, SOCK_STREAM, 0);
    bind(base_sd, (struct sockaddr *)(&my_addr),
10       sizeof(struct sockaddr_in));
```

Alla riga 1, si dichiara una variabile di tipo `struct sockaddr_in`. Alla riga 4, si azzera il contenuto della struttura, invocando la funzione di utilità `bzero()` la quale prende in ingresso l'indirizzo della struttura e la lunghezza in byte della struttura stessa.

Alla riga 5 si imposta la famiglia di protocolli che intendiamo utilizzare (protocolli internet). Alla riga 6, si specifica la porta su cui il server intende ricevere delle richieste di connessione. La variabile `myport` è un intero senza segno a 16 bit. La funzione di utilità `htons()` serve per convertire la convenzione di ordinamento dei byte da *host* a *network*[2]

Alla riga 7 si imposta l'indirizzo IP della struttura pari alla *wild card* `INADDR_ANY`. Infine, alla riga 8 viene invocata la `socket()` per creare il descrittore di socket che verrà utilizzato in seguito; e alla riga 9 viene invocata la bind per *legare* il socket appena create a un *end-point* TCP. Notare che nel prototipo della bind, il tipo del secondo parametro è `struct sockaddr *`, mentre nel nostro programma noi abbiamo preparato una struttura di tipo `struct sockaddr_in`. Questa differenza di tipo si deve al fatto che la `bind()` è una funzione generica in grado di lavorare con diversi protocolli (e non soltanto con i protocolli della famiglia Internet), e quindi è in grado di accettare una struttura generica; noi invece stiamo specificando il protocollo TCP/IP e quindi dobbiamo riempire la specifica struttura `struct sockaddr_in`. Le due strutture sono simili, e in particolare, in entrambe le strutture il primo campo specifica la famiglia di protocolli. Quindi la `bind()`, internamente accetta una struttura `struct sockaddr`, legge il primo campo, ed effettua una conversione di tipo (*casting*) a seconda della famiglia di protocolli.

Dal punto di vista dell'utente, dobbiamo fare una conversione di tipo da `struct sockaddr_in` a `struct sockaddr` ogni volta che invochiamo la `bind()` per evitare warning o addirittura errori del compilatore.

Funzione listen() .

Dopo aver invocato la bind, il server è quasi pronto per accettare richieste. Manca un ulteriore passo: l'invocazione della funzione `listen()`.

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

int listen(int sockfd, int backlog);
```

[2] Certi host assumono la convenzione xxx, mentre certi altri assumono la convenzione xxx nello stabilire l'ordinamento dei byte in una parola a 16 o a 32 bit. La funzione `htons()` converte la convenzione dell'host a quella del network. In certi host, quindi, tale funzione non fa assolutamente niente, mentre in altri host scambia l'ordine dei byte.

Argomenti in ingresso:

sockfd descrittore del socket (valore di ritorno della funzione `socket()`

backlog numero massimo di richieste di connessione che si possono accodare prima che vengano processate dalla `accept()`.

La chiamata alla listen ha due scopi:

- Per prima cosa, comunica al sistema che il server vuole ricevere richieste di connessione su socket specificato. In gergo, si dice che il socket `sockfd` viene trasformato in un socket *passivo*. Senza la listen, il sistema operativo non potrebbe accettare richieste di connessione dai client.
- Specificare la lunghezza della coda di richieste. Ogni richiesta da parte di un client viene inizialmente accodata in una coda di richieste *incomplete*, in attesa che il 3-way handshake venga completato. Una volta completato, la richiesta si sposta in una coda di richieste completate in attesa che il server invochi la funzione `accept()` e processi la richiesta. Se il server è molto impegnato a servire delle richieste, può passare un po di tempo tra una chiamata e l'altra della `accept`. Durante questo intervallo, le richieste che arrivano da parte di altri client vengono accodate in una coda del sistema operativo, la cui lunghezza è specificata dal paramtero `backlog`. Per la precisione, il parametro `backlog` rappresenta la somma delle lunghezze massime della coda delle richieste incomplete e della coda delle richieste complete.

In molti esempi riportati nei libri di testo, il valore tipico del parametro `backlog` è 5. In realtà, nei moderni server questo valore è totalmente inadeguato (per esempio, nei web-server moderni ci possono essere milioni di connessioni al giorno). Un tipico valore utilizzato oggi è 64. Notare che più alto è tale valore, maggiore è la quantità di memoria che il sistema operativo deve riservare per le code interne. Quindi, è opportuno settare tale valore dipendentemente dal tipo di programma server e dall'utilizzo che si aspetta di farne.

Se le code sono piene, e un nuovo pacchetto di richiesta di nuova connessione (SYN) arriva dalla rete, il pacchetto viene semplicemente scartato. Il protocollo TCP garantirà che dopo un certo intervallo (timeout), il pacchetto SYN verrà inviato nuovamente dal client, nella speranza che le code si siano nel frattempo svuotate.

Notare che invece, se nessun server è in ascolto su una certa porta, il sistema operativo normalmente risponde a una richiesta di connessione con un paccheto RST, e la funzione connect del client ritornerà con un errore.

Funzione accept() Dopo aver chiamato le tre funzioni `socket()` `bind()` e `listen()`, il sistema operativo è in ascolto sulla porta specificata per richieste di nuove connessioni. Per accettare una richiesta di connessione, il server deve invocare la funzione bloccante `accept()`.

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

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Argomenti in ingresso:

sockfd descrittore del socket passivo su cui si accettano richieste;

cliaddr al ritorno della funzione contiene l'indirizzo internet (IP + porta) del client che ha effettuato la richiesta;

addrlen al ritorno della funzione, contiene la lunghezza in byte della struttura cliaddr tornata come primo parametro.

Valore di ritorno:

- La funzione ritorna un *nuovo* descrittore di socket che può essere utilizzato per scambiare dati con il client.
- In caso di errore, ritorna il valore -1.

La funzione `accept()` quindi genera un nuovo descrittore di socket. Quest'ultimo è quello che viene effettivamente utilizzato per comunicare con il client. Quindi, un server tipicamente utilizza un socket unico, detto *socket passivo*, che esiste per tutta la durata del programma e che server per accettare nuove connessioni da parte dei client; e un socket diverso per ogni nuova connessione, che ha un tempo di vita pari alla durata della connessione.

E' importante sottolineare che nel protocollo TCP, una connessione è univocamente determinata dalla coppia di due end-point: l'indirizzo internet (IP + porta) del client e l'indirizzo internet (IP + porta) del server. In particolare, ci saranno più connessioni (una per ogni client) che avranno lo stesso end-point (quello del server).

Funzione connect Dopo aver visto come funziona il server, vediamo ora come funziona il client: la sequenza di operazioni da compiere è molto più semplice, e consiste nella chiamata delle funzioni `socket()` e `connect()`.

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

int connect(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);
```

Argomenti in ingresso:

sockfd il descrittore di socket da usare per la connessione;

saddr puntatore a una struttura contenente l'indirizzo internet del server;

addrlen lunghezza in byte della struttura di cui al secondo parametro.

Valore di ritorno:

- 0 se la connessione è andata a buon fine;
- -1 in caso di errore.

Notare che il client non ha bisogno di chiamare `bind` per specificare il proprio indirizzo, in quanto l'indirizzo IP è automaticamente stabilito dal nodo su cui il programma sta girando, mentre la porta viene scelta casualmente dal sistema operativo fra una delle porte libere di tipo *ephemeral*.

In seguito alla chiamata a `connect()` il protocollo TCP comincia il 3-way handshake che potrebbe portare alla connessione. Ci sono vari tipi di problemi che possono impedire lo stabilirsi di una connessione:

- Se non ci sono server in ascolto sulla porta specificata dal client, allora molto probabilmente il sistema operativo sul nodo destinazione manda un pacchetto RST in risposta al pacchetto SYN inviato dal client. Questo risulta in un errore di tipo `ECONNREFUSED`.
- Se non si riesce a raggiungere l'host destinazione, i router intermedi si scambieranno dei pacchetti ICMP, e alla fine verà ritornato un errore di `EHOSTUNREACH`, oppure di `ENETUNREACH`.
- Infine, se il server si rifiuta di rispondere al pacchetto SYN iniziale (perchè le code sono piene, oppure per esplicita configurazione del sistema operativo dell'host di destinazione), dopo un po (tipicamente 75 secondi) la connect ritorna con un errore di tipo `ETIMEDOUT`, a significare che nonostante sia possibile raggiungere l'host, e nonostante le ritransmissioni, l'host destinazione non ha mai risposto.

Chiusura della connessione E' possibile chiudere la connessione sia per il client che per il server semplicemente invocanto la funzione `close()`.

```
#include <unistd.h>

int close(int fd);
```

La funzione è la stessa che si usa per chiudere i descrittori di file, di pipe e di FIFO. Nel caso del socket di tipo Internet, la chiamata alla close comporta la chiusura della connessione remporta. Il socket viene marcato come *chiuso*, e si ritorna immediatamente al programma. Contemporaneamente, l'impementazione del protocollo TCP/IP all'interno del sistema operativo si occuperà di continuare a inviare i dati accodati nelle code interne che non sono ancora stati inviati, e quindi di chiudere la connessione inviando i pacchetti necessari.

Notare anche che, nel caso in cui il descrittore di socket sia condiviso fra più processi, in seguito alla chiamata alla close() viene decrementato un *reference counter* che tiene traccia del numero di processi concorrenti il cui descrittore di socket è aperto. Solo quando tale numero è pari a zero, la chiusura viene effettivamente fatta.

7.2.2. Esempio

In questa sezione facciamo un esempio completo di programma client server. Il nostro semplicissimo server sequenziale si occupa di leggere una stringa inviategli dal client,

trasformare tutti i caratteri in maiscoli, e rispedirli indietro al client. Nei programmi seguenti, faremo uso di una serie di funzioni di utilità, contenute nell'header m̈yutils.hc̈he viene riportato in appendice. Cominciamo a descrivere il client.

```
1  #include <sys/types.h>
   #include <netinet/in.h>
3  #include <sys/socket.h>
   #include <netdb.h>
5
   #include "myutils.h"
7
   #define BUFFERSIZE   2000
9
   struct sockaddr_in s_addr;
11 struct hostent     *server;

13 int  sd, s_port;
   char msg[100];
15
   int main(int argc, char *argv[])
17 {
     if (argc < 3) user_err("usage: client <servername> <port>");
19
     s_port = atoi(argv[2]);
21
     sd = socket(AF_INET, SOCK_STREAM, 0);
23
     bzero(&s_addr, sizeof(struct sockaddr_in));
25   s_addr.sin_family = AF_INET;
     s_addr.sin_port = htons(s_port);
27   server = gethostbyname(argv[1]);
     if (server == 0) sys_err("server not found!");
29
     bcopy((char*)server->h_addr,
31         (char*)&s_addr.sin_addr.s_addr,
           server->h_length);
33
     if (connect(sd, CAST_ADDR(&s_addr), sizeof(struct sockaddr_in)) < 0)
35     sys_err("connect failed!");

37   sprintf(msg, "I am the client with PID n %d\n", getpid());
     printf("Client: sending message!\n");
39   write(sd, msg, strlen(msg));
     read(sd, msg, 100);
41   printf("client: Message received back: %s", msg);
     close(sd);
43 }
```

Alla riga 10, dichiariamo la variabile `s_addr` che conterrà l'indirizzo del server. Alla riga 11, dichiariamo il puntatore `server` a una struttura dati `struct hostent` che

conterrà l'indirizzo del server.

Nel nostro programma, l'indirizzo simbolico o numerico del server e la porta sulla quale connettersi vengono passati al programma client sulla riga di comando. In particolare, `argv[1]` deve contenere l'indirizzo numerico o alfabetico del server, mentre `argv[2]` contiene la porta. La riga 20 serve appunto per ottenere il valore numerico della porta. Alla riga 22, creiamo il socket, mentre alle righe 24-26 cominciamo ad inizializzare l'indirizzo del server. Alla riga 27, invochiamo la funzione `gethostbyname()` che trasforma una stringa contenente un indirizzo simbolico o numerico in una struttura `struct hostent`:

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

As esempio, l'indirizzo internet ẅww.sssup.itv̈iene tradotto in un indirizzo IP numerico che verrà inserito nel campo `h_addr` della struttura `struct hostent` che viene ritornata dalla funzione.

Attenzione: `gethostbyname()` ritorna un puntatore a una zona di memoria definita staticamente. Se la funzione viene invocata più volte, la zona di memoria statica viene ogni volta sovrascritta. Quindi, invocando un pezzo di codice come il seguente:

```
struct hostent *s1,*s2;
...

s1 = gethostbyname("www.sssup.it");
s2 = gethostbyname("www.ing.unipi.it");
```

sia la variabile `s1` che la variabile `s2` puntano alla stessa zona di memoria, e quindi conterranno lo stesso indirizzo! In pratica, la seconda chiamata alla `gethostbyname` ha semplicemente sovrascritto la prima.

Anche una semplice copia della struttura potrebbe non bastare, in quanto la struttura contiene dei puntatori, e copiare dei puntatori non è purtroppo sufficiente (bisogna copiare tutti i campi uno per uno). Fare molta attenzione quindi all'uso di questa funzione!

La funzione `gethostbyname` ritorna il valore 0 se non riesce a stabilire la corrispondenza fra simbolico e numerico. Tale corrispondenza viene realizzata tramite un opportuno protocollo e una serie di strumenti, come il *name server*, una entità presente sulla rete locale che si occupa di effettuare tali corrispondenze.

Alla riga 30, copiamo l'indirizzo ottenuto (contenuto nel campo `h_addr` della struttura `server`), nel campo `s_addr`.

Alla riga 34, effettuiamo la connessione al server. Se va a buon fine, inviamo il messaggio da convertire con una semplice `write()` (riga 39), e aspettiamo la risposta del server con una `read()`. Una volta ricevuta la risposta, la stampiamo sullo schermo, chiudiamo il descrittore del socket (terminando la connessione), e terminiamo il programma.

Il codice del server è riportato qui di seguito.

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include "myutils.h"

#define BUFFERSIZE    2000

int init_sd(int myport);
void do_service(int sd);

int main(int argc, char *argv[])
{
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen = sizeof(struct sockaddr_in);
  int myport;
  int err = 0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);

  while (!err) {
    curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    if (curr_sd < 0) sys_err("accept failed!");

    do_service(curr_sd);
    close(curr_sd);
  }
  close(base_sd);
}

int init_sd(int myport)
{
  struct sockaddr_in my_addr;
  int sd;

  bzero(&my_addr, sizeof(struct sockaddr_in));
  my_addr.sin_family = AF_INET;
  my_addr.sin_port = htons(myport);
  my_addr.sin_addr.s_addr = INADDR_ANY;

  sd = socket(AF_INET, SOCK_STREAM, 0);
  if (bind(sd, CAST_ADDR(&my_addr), sizeof(my_addr)) < 0)
    sys_err("bind failed!");

  if (listen(sd, 5) < 0)
    sys_err("listen failed!");
```

```
50    printf("Server listening on port %d\n", myport);

52    return sd;
}
54
void do_service(int sd)
56{
    int i, l=1;
58  char msg[BUFFERSIZE];
    char ris[BUFFERSIZE];
60
    do {
62    l = read(sd, msg, BUFFERSIZE - 1);
      if (l == 0) break;
64    msg[l] = 0;
      printf("Server: received %s\n", msg);
66    for (i=0; i<l; i++) msg[i] = toupper(msg[i]);
      printf("Server: sending %s\n", msg);
68    write(sd, msg, l);
    } while (l!=0);
70}
```

Per strutturare meglio il codice, definiamo due funzioni intermedie in cui racchiuderemo parte del codice del server. La funzione `init_sd()` si occupa di inizializzare il socket su cui il server accetterà le richieste, mentre la funzione `do_service()` si occupa di realizzare il servizio per un client (nel nostro caso, rendere in caratteri maiuscoli una stringa di testo). I prototipi di tali funzioni sono dichiarati alle righe 8 e 9.

Il programma prende in ingresso come parametro (`argv[1]`) sulla linea di comando il numero di porta su cui il server si metterà in ascolto, e lo assegna alla variabile `myport` alla riga 20. Quindi viene chiamata la funzione `init_sd` il cui corpo sta alle righe 34-53. L'indirizzo internet del server viene inizializzato nelle righe 39-42. Quindi viene eseguita la sequenza di chiamate `socket()` – `bind()` – `listen()` propria di ogni server alle righe 44-48. A questo punto il server è in grado di ricevere richieste da parte del client, l'inizializzazione è terminata, e la funzione restituisce il descrittore di socket `sd`.

Il server (nel main) entra quindi in un ciclo `while` infinito alla riga 24, in cui prima esegue la primitiva `accept()` per ottenere un descrittore di socket su cui effettuare il dialogo con il client; quindi invoca la funzione `do_service()` che si occupa di effettuare il servizio e comunicare la risposta al client.

Quindi, chiude il descrittore di socket temporaneo `curr_sd`, e si rimette in attesa di una richiesta sul descrittore `base_sd` tramite una nuova chiamata alla `accept()`.

7.3. Server concorrenti

Nell'esempio di server fatto nella sezione precedente, il server serve le richieste una alla volta, nell'ordine di arrivo (FIFO). Quindi, un cliente che effettui una richiesta deve aspettare il completamento di tutte le richieste precedenti, e nel caso che il tempo di

servizio di una richiesta sia variabile (per esempio richieste che richiedono tempi di servizio lunghi e richieste che richiedano tempi di servizio brevi), i client con richieste brevi possono dover aspettare per molto tempo che tutte le richieste lunghe precedenti siano terminate.

Questo è particolarmente vero nel caso in cui il servizio contenga un certo numero di accessi di I/O. Ad esempio, un server web serve richieste che nella maggior parte dei casi richiedono la lettura di uno o più file da disco fisso. Poiché le operazioni di I/O sono in generale più lente delle operazioni CPU bound, un server strettamente sequenziale rischia di non utilizzare al massimo le risorse del sistema.

Sarebbe opportuno, invece, poter sfruttare lo schedulatore del sistema operativo per poter gestire le richieste concorrentemente, nell'ottica di aumentare il *throughput*, ovvero la quantità di richieste che è possibile servire in media nell'unità di tempo.

Ci sono varie tecniche per implementare un server concorrente. Nel seguito, ne descriveremo alcune.

7.3.1. Server multiprocesso

In questo tipo di approccio, si usa un processo distinto per ogni richiesta. Quindi, si esegue una fork di un nuovo processo figlio ogni volta che si riceve una nuova richiesta.

Il codice del programma è riportato di seguito.

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include "myutils.h"


#define BUFFERSIZE    2000

int init_sd(int myport);
void do_service(int sd);
void sig_child(int signo);

int main(int argc, char *argv[])
{
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen;
  int myport;
  int err = 0;
  int ch=0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);
```

```
28
   base_sd = init_sd(myport);
30
   signal(SIGCHLD, sig_child);
32
   while (!err) {
34   if ( (curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen)) < 0) {
       if (errno == EINTR)
36       continue;
       else sys_err("accept failed!");
38   }
     ch = fork();
40   if (ch == 0) {
       do_service(curr_sd);
42     close(curr_sd);
       exit(0);
44   }
     close (curr_sd);
46 }
   close(base_sd);
48 }

50 void sig_child(int signo)
   {
52   pid_t pid;
     int stat;
```

Il codice delle funzioni `init_sd()` e `do_service()` non è riportato perché identico a quello dell'esempio precedente (server sequenziale).

Alla linea 33, invece di invocare direttamente la `do_service()`, effettuiamo una fork e lasciamo il lavoro al processo figlio, mentre il processo padre si rimette in attesa sulla accept. Quando il processo figlio termina, un segnale di tipo `SIGCHLD` viene inviato al processo padre. Per evitare processi zombie, alla ricezione di questo messaggio viene invocato l'handler `sig_chld()` (che è stato installato alla linea 27) che si occupa di chiamare la `waitpid()` per recuperare le informazioni sui processi appena terminati.

Notare che tale segnale potrebbe interrompere il processo padre mentre esso è bloccato sulla primitiva bloccante `accept()`, e quindi tale primitiva ritornerebbe con errore. Per distinguere tale caso da un errore diverso, alla linea 31 controlliamo il contenuto della variabile `errno` e se è pari a `EINTR`, allora la `accept` è stata interrotta dal segnale e quindi semplicemente ci rimettiamo in attesa ricomincindo il ciclo. Altrimenti, usciamo con errore.

7.3.2. Server multithread

In alternativa ai processi, possiamo creare un thread ogni volta che arriva una richiesta. L'esempio precedente si modifica come segue.

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

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <pthread.h>
#include "myutils.h"

#define BUFFERSIZE    2000

int init_sd(int myport);
void do_service(int sd);

void *body(void *arg)
{
  int sd = *((int*) arg);
  int i,l;

  free(arg);
  pthread_detach(pthread_self());
  do_service(sd);
  close(sd);
}

int main(int argc, char *argv[])
{
  pthread_t tid;
  struct sockaddr_in c_add;
  int base_sd, curr_sd;
  int addrlen;
  int myport;
  int err = 0;

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);

  base_sd = init_sd(myport);

  while (!err) {
    curr_sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    if (curr_sd < 0) sys_err("accept failed!");
    int *pointer = malloc(sizeof(int));
    *pointer = curr_sd;
```

Notare l'uso della primitiva `pthread_detach()` per impedire la creazione di thread zombie.

La creazione di un thread per ogni richiesta, sebbene notevolmente più leggera della creazione di un processo per ogni richiesta, porta comunque un certo overhead che in certi casi può essere necessario eliminare. Una possibile tecnica è quella di creare una serie di thread pronti in attesa di servire una richiesta.

Tali thread potrebbero essi stessi chiamare la primitiva `accept()`, eliminando la

necessità di un thread principale di coordinamento. Non è però possibile per più di un thread bloccarsi sulla `accept()` su uno stesso socket, in quanto in caso di arrivo di una richiesta il sistema operativo non capirebbe quale thread sbloccare. Inoltre, le strutture dati interne alla coda delle richieste vanno protette da una eventuale accesso simultaneo.

Il codice risultante è mostrato di seguito.

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <pthread.h>
#include "myutils.h"

#define BUFFERSIZE    500
#define MAXNTHREAD    5

pthread_t tid[MAXNTHREAD];
pthread_mutex_t m_acc;

int init_sd(int myport);
void do_service(int sd);

void *body(void *arg)
{
  struct sockaddr_in c_add;
  int addrlen;
  int i,l;
  int base_sd = *((int *) arg);
  int sd;

  while (1) {
    pthread_mutex_lock(&m_acc);
    sd = accept(base_sd, CAST_ADDR(&c_add), &addrlen);
    pthread_mutex_unlock(&m_acc);

    do_service(sd);
    close(sd);
  }
}

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

  if (argc < 2) user_err("usage: server <port>");
  myport = atoi(argv[1]);
```

```
44    base_sd = init_sd(myport);
      pthread_mutex_init(&m_acc, 0);
46
      for (i=0 ; i<MAXNTHREAD; i++)
48      pthread_create(&tid[i], 0, body, &base_sd);

50    pause();
  }
```

In pratica, si dichiara un mutex che serve per evitare che più di un thread effettui la chiamata alla `accept()`. Per il resto, il main si limita a inizializzare le strutture, a creare i thread, e quindi ad addormentarsi indefinitivamente,.

Esistono naturalmente altri modi per strutturare un server concorrente. É indubbio che l'uso dei thread, oltre a rendere il programma più leggibile, riduca notevolmente l'overhead.

A. Funzioni di utilità

Riportiamo qui di seguito il contenuto del file di utilità m̈yutils.h:

```
#ifndef __MYUTILS_H__
#define __MYUTILS_H__

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

#define CAST_ADDR(x) (struct sockaddr *)(x)

void sys_err(char *s)
{
  perror(s);
  exit(-1);
}

void sys_warn(char *s)
{
  perror(s);
}

void user_err(char *s)
{
  printf("%s\n", s);
  exit(-1);
}

#endif
```

Indice analitico

_exit, 9
accept, 92
alarm, 47
atexit, 9
bind, 89
chmod, 22
close, 94
connect, 93
dup, 19
execve, 32
exit, 9
fdopen, 24
feof, 23
fflush, 23
fgets, 24
fopen, 22
fork, 27
get, 58
getchar, 23
getenv, 10
gethostbyname, 96
kill, 42
listen, 91
lseek, 17
main, 8
malloc, 12
mkfifo, 55
open, 16
pause, 43
pipe, 50
pthread_attr_init, 70
pthread_attr_setschedpolicy, 71
pthread_cancel, 85
pthread_cleanup_push, 85
pthread_cond_wait, 77
pthread_create, 67
pthread_detach, 68
pthread_join, 67
pthread_kill, 68
pthread_mutex_init, 73
pthread_mutex_lock, 73
pthread_mutex_trylock, 73
pthread_setconcurrency, 66
putc, 23
sem_getvalue, 82
sem_init, 81
sem_trywait, 82
sem_wait, 81
setenv, 10
sigaction, 38
signal, 37
sigsuspend, 45
sleep, 43
socket, 87, 88
wait, 30

pthread_mutex_t, 72
pthread_t, 66

sec:kill, 41
segnali, 36
sem_t, 81
signal handler, 37

Thread:attesa di condizione, 77
Thread:attributes, 70
Thread:creazione, 66
Thread:mutua esclusione, 72

Bibliografia

[1] Richard W. Stevens. *Unix Network Programming.* Prentice-Hall, 1998.