Programmazione concorrente su sistemi Unix

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

Capitolo 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 Note generali su Unix

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*). Tali syscall dipendono dal sistema operativo, ed in questa sede faremo riferimento ai sistemi operativi Unix-like (Linux in particolare), secondo lo standard POSIX.

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.

```c
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.**

Capitolo 2

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.

2.1 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

2.1.1 Identificatori di processo

Come detto, ogni processo Unix è caratterizzato da un PID privato, che è l'entità con cui il sistema identifica tale processo. 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 2.1.5 il perché di questo comportamento.

2.1.2 Creazione di nuovi processi

È possibile creare un nuovo processo tramite la primitiva `fork()`, che crea un processo "figlio" del processo che la invoca, assegnandogli un nuovo PID e copiando nello stato del processo figlio e nelle sue strutture private lo stato e le strutture private del padre. Il processo figlio avrà quindi un proprio spazio di memoria privato, il quale contiene una copia (sia per quanto concerne i dati che per quanto concerne il codice) dello spazio di memoria del padre. Per questo motivo il figlio "vedrà" tutte le variabili col valore che avevano nel padre prima dell'esecuzione della `fork()`, ma potrà modificare solo la propria copia locale delle variabili.

La sintassi della `fork()` è la seguente:

```
#include <unistd.h>

pid_t fork(void);
```

Sia padre che figlio (in caso di corretta terminazione della syscall) continueranno la loro esecuzione da dopo la `fork()` (ricordiamo che il corpo del figlio è una copia del corpo del padre): tale syscall verrà quindi chiamata una volta e ritornerà due volte (una nel processo padre ed una nel processo figlio). 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;
...
res = fork();
if (res < 0) {
    perror("fork failed");
    exit(-1);
}

if (res == 0) {
    /* We are in the child process */
    ...
}
else {
    /* We are in the father process */
    ...
}
```

Come si vede, una soluzione di questo tipo non è però 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 */
    ...
```

```
  exit(1);
}

/* We are in the father process */
...
```

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

2.1.3 Caricamento di un programma in memoria

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.

Tale problema può essere risolto sostituendo il corpo del processo figlio dopo la `fork()` (caricando nello spazio di memoria del figlio un nuovo codice e dei nuovi dati). Per permettere questo, Unix mette a disposizione la famiglia di funzioni `exec()`, che sovrascrive 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). Parlando 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[])
```

<u>Argomenti in ingresso</u> :

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) {
        perror("execve failed");
        exit(-1);
    }

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

2.1.4 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 : Perche' la prima printf del processo padre
 * a volte avviene per due volte????
 * Spiegare il comportamento quando invece l'output
 * e' rediretto su file .....
 */

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

    /* A questo punto parto con la fork() */
    if ((child = fork()) < 0) { perror("Errore nella fork");
        exit(-1);
    }
    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.
         */
        /*if (execlp("figlio", "figlio", "arg1", "arg2") < 0){*/
        if (execvp("figlio", array) < 0){
```

```
            perror("E' fallita la exec");
            exit(-1);
        }
    }
    else {
        /* Sono nel processo padre */
        printf("Processo padre ....  pid_t = %d\n", getpid());
        sleep(1);
    }
    return 0;
}

/* Programma che viene eseguito tramite la primitiva
 * exec. Si limita a stampare a video i parametro che gli
 * sono passati in input
 */
#include <stdio.h>
#include <unistd.h>

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

    printf("Programma 'execed' - Processo padre = %d\n",
getppid());
    for (i=0; i<argc; i++) {
        printf("Argomento %d = %s \n", i, argv[i]);
    }
    return 0;
}
```

2.1.5 Terminazione dei processi

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

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 percé il flusso di esecuzione è arrivato alla fine della funzione `main()` o perché essa ha effettuato una `return`). Questo significa che il runtime C chiama la funzione `main()` all'incirca in questo modo:

```
/* Preparazione di argv e argc */
```

res = main(argc, argv);
exit(res);

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);
```

<u>Argomenti in ingresso</u> :

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

<u>Valore restituito</u> :

-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 2.2.1. 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 2.2.1, 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()`.

2.2 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; per produrre un lavoro utile, i processi che compongono un programma concorrente devono però sincronizzarsi e comunicare in maniera meno banale. In questa sezione vedremo come vari processi possono sincronizzarsi fra loro tramite `signal` e scambiarsi dati in maniera sincrona o asincrona.

2.2.1 I segnali

La sincronizzazione fra processi cooperanti può essere ottenuta permettendo ad alcuni processi di attendere eventi asincroni generati da altri, sincronizzandosi con essi. I *segnali* sono il meccanismo utilizzato dai sistemi Unix per rendere possibile tale tipo di sincronizzazione: in pratica, un segnale è un evento asincrono, talvolta designato col nome di interrupt software, che un processo può *inviare* ad un altro processo (o ad un gruppo di processi). 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 una delle tre azioni sopra è detto *pendente*.

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

- specificare l'azione da intraprendere in risposta ad un segnale,
- bloccare un segnale (mascherarlo),
- controllare se ci sono segnali pendenti,
- inviare segnali ad un altro processo (o far scattare segnali dopo un certo tempo).

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.

2.2.2 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>

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

int main(void)
{
    ...
    /* Installo l'handler e controllo il risultato */
```

```
    if (signal(SIGUSR1, handler) == SIG_ERR) {
        perror("Errore nell'allocazione delle SIGUSR1 \n");
        exit(-1);
    }
    ...
}
```

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, conviene utilizzare la syscall `sigaction()`, che ha un comportamento standard.

```
#include <signal.h>

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

Argomenti in ingresso :

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

act – Puntatore alla struttura contenente le informazioni riguardanti il 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.

sa_mask – Maschera dei signal 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>

/* 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) {
        perror("Errore nell'allocazione delle SIGUSR1 \n");
        exit(-1);
    }
    ...
}
```

Nell'esempio sopra è introdotta la funzione `sigemptyset()`. Questa funzione fa parte di tutta una famiglia di funzioni che servono per la gestione dell'annidamento dei `signal`. Può capitare, infatti, che scatti un signal mentre il task sta eseguendo l'handler relativo ad un altro. In questo caso il programmatore ha la scelta, come per le interrupt hardware, di servire il nuovo signal e quindi ritornare al vecchio, oppure ignorare il nuovo signal scattato fino a che non ha finito di servire quello attuale.

Si può specificare tale comportamento per ogni handler installato e questo e' fatto tramite tutta una famiglia di funzioni di cui la `sigemptyset()` è solo uno degli appartanenti. Tutte queste 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`.

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'em interfaccia del tipo. In particolare, l'interfaccia della `struct sigset_t` è la seguente:

```
#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 signal siano mascherate.

sigfillset – Modifica la struttura `set` in modo che tutti i signal siano attivi.

sigaddset – Aggiunge alla struttura `set` il signal `signum`, ossia lo rende attivo.

sigdelset – Maschera il signal `signum` nella struttura `set`.

sigismember – Controlla se il signal `signum` è attivato o mascherato nella struttura `signum`.

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 signal `signum` è attivato nella struttura `set`.

0 – Se il signal `signum` è mascherato nella struttura `set`.

-1 – In caso di errore.

2.2.3 Un esempio

In questa sezione esamineremo un seplice esempio di uso delle signal. Il programma crea due task. Alternativamente ognuno dei due task 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.

```
/* Breve esempio di uso di fork() e signal() che genera
 * due task che si scambiano alternativamente un signal
 * a intervalli "regolari di tempo.
 * Quesiti:
 * 1) Perche' e' stato inseria la linea
 * if (seconds == 0)
 * 2) Perche' anche con questa linea c'e' un caso
 * critico di deadlock e quale e'?
 */

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

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

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

int seconds;

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

    seconds = 0;
    /* Utilizzo la sintassi Posix per settare un handler
     */

    /* Inizializzo i campi della struttura sigaction */
    /* indirizzo dell'handler */
    nuova.sa_handler = handler;
    /* Quando l'handler scatta maschera TUTTI i signal */
    sigemptyset(&nuova.sa_mask);
    /* Nessun flag viene settato */
    nuova.sa_flags = 0;

    /* Installo l'handler prima della fork in modo che
     * sia ereditato dal processo figlio!!!
     */
    sigaction(SIGUSR1, &nuova, &vecchia);

    if ((child = fork()) < 0) {
        perror("Errore nella fork ....  \n");
        exit(-1);
    }
    else if (child == 0) {
        /* A questo punto sono nel task figlio */

        /* Ottengo il pid del padre */
        father = getppid();
        for(;;) {
            sleep(1);
```

```
            kill(father, SIGUSR1);
            pause();
            printf("Secondi trascorsi (figlio) %d -- time =
%d\n", 2*seconds,
                         (int)time(NULL));
        }
    }
    else {
        if (seconds == 0)
            pause();
        /* A questo punto sono nel padre */
        for(;;){
            printf("Secondi trascorsi (padre) %d -- time = %d
\n", 2*seconds-1,
                         (int)time(NULL));
            sleep(1);
            kill(child, SIGUSR1);
            pause();
        }

    }
}

/* Handler del signal */
static void handler(int signo)
{
    seconds++;
}
```

2.3 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.

2.3.1 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]);
```

<u>Argomenti in ingresso</u> :

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) {
    perror("Fork error");
    exit(-1);
}
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`.

2.3.2 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>

/* 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) {
        perror("Errore nell'installazione della SIGPIPE");
        exit(-1);
    }
    printf("Lunghezza del buffer della FIFO %ld\n",
                sysconf(_PC_PIPE_BUF));

    if (pipe(pipefd) == -1) {
        /* C'e' stato un errore nell'apertura della pipe */
        perror("Errore nella creazione della pipe");
        exit(-1);
    }

    if ((child = fork()) < 0) {
        perror("Errore nella fork");
        exit(-1);
    }
    else if (child == 0) {
        /* Il processo figlio chiude il descrittore
         * in lettura poiche' 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) {
                perror("Errore nella write ...");
```

```
                exit(-1);
            }
        }
        /* Come fine stringa invio un byte a 0 */
        if (write(pipefd[1], 0, sizeof(char)) == -1) {
            perror("Errore nella write");
            exit(-1);
        }
        printf("Fine della scritta\n");
        close(pipefd[1]);
    }
    else {
        /* Il processo padre chiude la pipe in scrittura
         * perche' deve solo leggere ....
         */
        close(pipefd[1]);
        /* Per vedere scattare la SIGPIPE provare a scommentare le 2
         * linee che seguono.
         * 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) {
                perror("Errore nella read ...");
                exit(-1);
            }
            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);
}
```

2.3.3 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`.

2.3.4 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)
```

<u>Argomenti in ingresso</u> :

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.

2.3.5 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).

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

#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>

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

/* Defizione e istanziazione di una struttura usata
 * dai due task per "giocare".
 */
struct tris {
    char campo[9];
    /* Numero di caselle libere */
    int 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 |
                                IPC_CREAT | IPC_EXCL);
    if (memd == -1) {
        perror("Non sono riuscito a creare la memoria
condivisa\n");
        exit(-1);
    }
    /* "attacco" la zona di memoria condivisa al mio spazio
     * di indirizzamento */
    mioCampo = shmat(memd, 0, SHM_R | SHM_W);
    if (mioCampo == -1) {
        perror("Non sono riuscito ad agganciare la memoria
condivisa\n");
        exit(-1);
    }

    /* Inizializzo la zona di memoria condivisa .... */
    for (i=0;i<9;i++)
        mioCampo->campo[i] = LIBERO;
    mioCampo->libere=9;
    /* Il padre e' quello che inizia a giocare per primo!!! */
    mioCampo ->turno = PADRE;

    /* A questo punto parto con le fork .... */
    if ((child = fork()) < 0) {
        perror("Errore nella fork");
        exit(-1);
    }
```

```
else if (child == 0) {
    /* Processo figlio .... */
    while(mioCampo→libere > 0) {
        /* Aspetto in polling che il turno passi al figlio */
        while(mioCampo→turno == PADRE);

        /* Estraggo caso una delle caselle libere */
        mossa = (rand()%mioCampo→libere)+1;

        /* Riempio la casella estratta prima con il simbolo
         * "giusto".
         */
        for (i=0; i<9; i++) {
            if (mioCampo→campo[i] == LIBERO) {
                mossa--;
                if (mossa == 0) {
                    mioCampo→campo[i] = 'o';
                    break;
                }
            }
        }
        /* "Pulisco" lo schermo */
        system("clear");
        /* Stampo a video il "campo" di tris */
        stampaCampo(mioCampo→campo);
        /* Aggiorno il valore delle caselle libere */
        mioCampo→libere--;
        /* Cedo il turno all'altro task */
        mioCampo→turno = PADRE;
        /* Fermo il task per 1 secondo */
        sleep(1);
    }
    /* In ogni caso prima di uscire libero il semaforo */
    mioCampo→turno = PADRE;
    return 0;
}
else {
    /* Processo Padre .... */
    while(mioCampo→libere > 0) {
        /* Aspetto in polling che il turno passi al padre */
        while(mioCampo→turno == FIGLIO);
```

```
        /* Estraggo caso una delle caselle libere */
        mossa = (rand()%mioCampo->libere)+1;

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

  }
  return 0;
}
```

Capitolo 3

Multithreading

Come già detto precedentemente, un thread è un flusso di esecuzione non necessariamente legato ad uno spazio di memoria privato (ed in questo si differenza da un processo). Un thread ha bisogno di uno spazio di memoria in cui eseguire, ma tale spazio non è necessariamente privato. In questo senso si può dire che ogni processo conterrà uno o più thread: 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*.

I thread creati da un processo girano nello stesso spazio di indirizzamento (quello del processo che li ha creati) e quindi possono usufruire, senza alcun aggravio da parte del kernel, di una zona di memoria condivisa tramite la quale può avvenire lo scambio di dati. Naturalmente la gestione della concorrenza fra thread resta un onere dell'applicazione.

3.1 Generalità sui Thread

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, Program Counter),
- Stack,
- Variabile *errno*,
- Maschera dei Signal,
- Priorità.

Come avviene sempre nella programmazione il supporto per determinate caratteristiche all'interno di un sistema operativo viene dato essenzialmente tramite due meccanismi: *tipi di dato* e *funzioni del sistema operativo*. Nei prossimi paragrafi introdurremo quindi con gradualità sia alcuni dei nuovi tipi di dato che alcune delle nuove funzioni implicate nella gestione dei thread.

3.1.1 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 deve comunicare al nucleo il numero di thread da creare detto *concurrency level*. Ciò è fatto invocando la routine :

```
#include <pthread.h>
int pthread_setconcurrency (int nThread);
```

Argomenti in ingresso :

nThread – Numero dei thread che si vogliono creare.

Valore restituito :

0 – In caso di successo.

≠ 0 – In caso di insuccesso.

Un thread è creato invocando la seguente routine del nucleo :

```
#include <pthread.h>

int pthread_create (pthread_t *tid, const p_thread_attr *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 i dati.

Valore restituito :

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 funzione di nucleo :

```
#include <pthread.h>

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

Argomenti in ingresso :

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

status – Puntatore all'area di memoria in cui viene salvato il valore retirtuito dal thread.

Valore restituito :

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.

La funzione **pthread_kill**, serve per mandare un segnale software a un thread. Il nome della funzione è giustificato dal fatto che in genere tale funzione viene usata per inviare un segnale di terminazione. Il problema delle segnalazioni software a un thread o a un task sarà affrontato più approfonditamente in seguito.

```
#include <pthread.h>

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

Argomenti in ingresso :

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

signo – Identificatore del segnale che si vuole inviare.

Valore restituito :

0 – In caso di successo.

$\neq$ **0** – In caso di insuccesso.

3.1.2 Un esempio: 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.

```
/* Programma di esempio di due thread .... */

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

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

#define CICLI 10
/* Variabile che regola il tempo perso nel ciclo for */
#define DELAY 2000000

/* Memoria Condivisa fra i thread */
struct {
    int val;
} shared;

/* Forward declaration */
void *somma1(void *);
void *somma2(void *);

int main(void)
{
    pthread_t som1TID, som2TID;
    int risu;

    /* Inizializzo la memoria condivisa */
    shared.val = 0;

    /* Setto il concurrency level a 2 */
    /*pthread_setconcurrency(2);*/

    /* A questo punto posso creare i thread .... */
    risu = pthread_create(&som1TID, NULL, somma1, NULL);
    if (risu != 0) {
        printf("Errore nella creazione del primo thread\n");
        return -1;
    }

    risu = pthread_create(&som2TID, NULL, somma2, NULL);
    if (risu != 0) {
        printf("Errore nella creazione del secondo thread\n");
        pthread_kill(som1TID, SIGKILL);
        return -1;
    }
```

```
    /* A questo punto aspetto che i due thread finiscano ... */
    pthread_join(som1TID, NULL);
    pthread_join(som2TID, NULL);

    printf("E' finito l'esperimento ....\n");
    return (0);
}

void *somma1(void *in)
{
    int i, j;

    for(i=0; i<CICLI; i++) {
        shared.val++;
        /* Ciclo messo solo per perdere tempo */
        for(j=0; j<DELAY; j++);
        printf("somma1 -- shared = %d \n", shared.val);
    }
    return NULL;
}

void *somma2(void *in)
{
    int i, j;

    for(i=0; i<CICLI; i++) {
        shared.val++;
        /* Ciclo messo solo per perdere tempo */
        for(j=0; j<DELAY; j++);
        printf("somma2 -- shared = %d \n", shared.val);
    }
    return NULL;
}
```

Dato che un **thread** altro non è che una funzione C che gira in maniera concorrente ad altre funzioni (nell'esempio i due thread sono le due funzioni **somma1** e **somma2**), 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 *somma1* e *somma2* accedono entrambi alla variabile condivisa *shared.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, poichè, 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.

3.2 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 routine di nucleo che vi operano.

Le routine 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 restituito :

0 – In caso di successo.

$\neq$ **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 restituito :

0 – Se il semaforo è stato acquisito.

EBUSY – Se il semaforo era già occupato.

È importante sottolineare che le routine e il tipo di dato appena visti servono solamente per casi di mutua esclusione. Per casi più complessi di sincronizzazione su una certa condizione esistono dei meccanismi più raffinati che verranno esaminati nel seguito.

3.2.1 Un esempio: Meccanismi di mutua esclusione

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

```
/* Programma di esempio di due thread ....
 * Ora aggiungiamo un semaforo di mutua esclusione!!!!
 */

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

/* Numero di cicli di scrittura/lettura nel buffer */
#define CICLI 10
/* Lunghezza del buffer */
#define LUN 50
/* Numero di cicli a vuoto di attesa */
#define DELAY 10000000

/* Memoria Condivisa fra i threads ...
```

```
 */
struct {
    pthread_mutex_t mutex;
    /* Stringa stampata a video */
    char scritta[LUN+1];
    /* Numero dei caratteri inseriti */
    int numLetture;
    int numScritture;
} shared = {PTHREAD_MUTEX_INITIALIZER};

/* Forward declaration */
void *scrittore1(void *);
void *scrittore2(void *);
void *lettore(void *);

int main(void)
{
    pthread_t scrit1, scrit2, let;
    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.numLetture = 0;
    shared.numScritture = 0;

    /* Setto il concurrency level a 3 */
    pthread_setconcurrency(3);

    /* A questo punto posso creare i thread .... */
    res = pthread_create(&scrit1, NULL, scrittore1, NULL);
    if (res != 0) {
        printf("Errore nella creazione del primo thread\n");
        return -1;
    }

    res = pthread_create(&scrit2, NULL, scrittore2, NULL);
    if (res != 0) {
```

```
        printf("Errore nella creazione del secondo thread\n");
        pthread_kill(scrit1, SIGKILL);
        return -1;
    }

    res = pthread_create(&let, NULL, lettore, NULL);
    if (res != 0) {
        printf("Errore nella creazione del secondo thread\n");
        pthread_kill(scrit1, SIGKILL);
        pthread_kill(scrit2, SIGKILL);
        return -1;
    }
    /* Fine della fase di creazione dei thread */

    /* A questo punto aspetto che i due thread finiscano ... */
    pthread_join(scrit1, NULL);
    pthread_join(scrit2, NULL);
    pthread_join(let, NULL);

    printf("E' finito l'esperimento ....\n");
    return (0);
}

void *scrittore1(void *in)
{
    int i, j, k;

    for (i=0; i<CICLI; i++) {
        while(1) {
            pthread_mutex_lock(&shared.mutex);
            if (shared.numLetture == shared.numScritture) {
                break;
            }
            pthread_mutex_unlock(&shared.mutex);
        }
        for(k=0; k<LUN; k++) {
            shared.scritta[k] = '+';
        }
        shared.numScritture++;
        pthread_mutex_unlock(&shared.mutex);
        for(j=0; j<DELAY; j++);
```

```
    }
    return NULL;
}

void *scrittore2(void *in)
{
    int i, j, k;

    for (i=0; i<CICLI; i++) {
        while(1) {
            pthread_mutex_lock(&shared.mutex);
            if (shared.numLetture == shared.numScritture) {
                break;
            }
            pthread_mutex_unlock(&shared.mutex);
        }
        for(k=0; k<LUN; k++) {
            shared.scritta[k] = '-';
        }
        shared.numScritture++;
        pthread_mutex_unlock(&shared.mutex);
        for(j=0; j<DELAY; j++);
    }
    return NULL;
}

void *lettore(void *in)
{
    int i;

    for (i=0; i<2*CICLI; i++) {
        while(1) {
            pthread_mutex_lock(&shared.mutex);
            if (shared.numLetture != shared.numScritture) {
                break;
            }
            pthread_mutex_unlock(&shared.mutex);
        }
        printf("Lettore -- scritta = %s\n", shared.scritta);
        shared.numLetture++;
        pthread_mutex_unlock(&shared.mutex);
```

```
    }
    return NULL;
}
```

L'esempio è una rivisitazione del classico problema dei lettori/scrittori. In questo caso si hanno due thread scrittori (**scrittore1** e **scrittore2**) 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.

Si noti come, visto che l'unico strumento che abbiamo è la mutua eclusione, la sincronizzazione sulla condizione di '*fine scrittura stringa*' può essere fatta solamemente tramite un polling.

Un'altra cosa da notare è che l'inizializzazione della struttura **mutex** è fatta in modo statico uguagliandola a una macro.

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.

3.3 Attesa di una condizione

Come detto nella sezione 3.2 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);
int pthread_cond_signal (pthread_cond_t *cond_var);
```

Argomenti in ingresso :

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

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

Valore restituito :

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, poichè 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.

3.3.1 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.

```
/* Programma di esempio di due thread ....
 * Ora aggiungiamo un semaforo di mutua esclusione!!!!
```

```
 * ... piu' una variabile condition!!!
 */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>

/* Numero di cicli di lettura/scrittura che vengono fatti dai thread */
#define CICLI 1
/* Lunghezza del buffer */
#define LUN 20
/* Numero di cicli di attesa a vuoto di uno scrittore */
#define DELAY_WRITER 200000
/* Numero di cicli di attesa a vuoto di uno scrittore */
#define DELAY_READER 2000000

/* Memoria Condivisa fra i thread ... */
struct {
    /* Semaforo di mutua esclusione */
    pthread_mutex_t mutex;
    /* Variabile condition per il lettore */
    pthread_cond_t lettore;
    /* Variabile condition per gli scrittori */
    pthread_cond_t scrittori;
    /* Buffer */
    char scritta[LUN+1];
    /* Variabili per la gestione del buffer */
    int primo, ultimo, elementi;
    /* Numero di lettori e scrittori bloccati */
    int blockscri, blocklet;
} shared = {PTHREAD_MUTEX_INITIALIZER,
                    PTHREAD_COND_INITIALIZER,
PTHREAD_COND_INITIALIZER};

/* Forward declaration */
void *scrittore1(void *);
void *scrittore2(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;
    shared.blocklet = shared.blockscri = 0;

    /* Setto il concurrency level a 3 */
    pthread_setconcurrency(3);

    /* A questo punto posso creare i thread .... */
    res = pthread_create(&lTID, NULL, lettore, NULL);
    if (res != 0) {
        printf("Errore nella creazione del primo thread\n");
        return -1;
    }

    res = pthread_create(&s1TID, NULL, scrittore1, NULL);
    if (res != 0) {
        printf("Errore nella creazione del secondo thread\n");
        /* Devo far terminare il thread che ho creato */
        pthread_kill(s1TID, SIGKILL);
        return -1;
    }

    res = pthread_create(&s2TID, NULL, scrittore2, NULL);
    if (res != 0) {
        printf("Errore nella creazione del terzo thread\n");
        pthread_kill(lTID, SIGKILL);
        pthread_kill(s1TID, SIGKILL);
        return -1;
    }

    /* A questo punto aspetto che i tre thread finiscano ... */
    pthread_join(s1TID, NULL);
```

```
    pthread_join(s2TID, NULL);
    pthread_join(lTID, NULL);

    printf("E' finito l'esperimento ....\n");
    return (0);

}

/* Codice relativo a uno dei thread che scrivono */
void *scrittore1(void *in)
{
    int i, j, k;

    for (i=0; i<CICLI; i++) {
        for(k=0; k<LUN; k++) {
            /* Come prima cosa acquisisco la mutua esclusione */
            pthread_mutex_lock(&shared.mutex);
            while (shared.elementi == LUN) {
                /* Segnalo che mi sto bloccando */
                shared.blockscri++;
                pthread_cond_wait(&shared.scrittori, &shared.mutex);
                /* Segnalo che mi sto sbloccando */
                shared.blockscri--;
            }
            /* Aggiungo un carattere e aggiorno i vari campi */
            shared.scritta[shared.ultimo] = '-';
            shared.ultimo = (shared.ultimo+1)%(LUN);
            shared.elementi++;
            /* Controllo se devo sbloccare il lettore */
            if (shared.blocklet ≠ 0)
                pthread_cond_signal(&shared.lettore);
            /* Rilascio la mutua esclusione */
            pthread_mutex_unlock(&shared.mutex);
            /* ... perdo tempo (in modo ORRIBILE!!!!) */
            for(j=0; j<DELAY_WRITER; j++);
        }
    }
    return NULL;
}

/* Codice relativo a uno dei thread che scrivono */
```

```
void *scrittore2(void *in)
{
    int i, j, k;

    for (i=0; i<CICLI; i++) {
        for(k=0; k<LUN; k++) {
            /* Come prima cosa acquisisco la mutua esclusione */
            pthread_mutex_lock(&shared.mutex);
            while (shared.elementi == LUN) {
                /* Segnalo che mi sto bloccando */
                shared.blockscri++;
                pthread_cond_wait(&shared.scrittori, &shared.mutex);
                /* Segnalo che mi sto sbloccando */
                shared.blockscri--;
            }
            /* Aggiungo un carattere e aggiorno i vari campi */
            shared.scritta[shared.ultimo] = '+';
            shared.ultimo = (shared.ultimo+1)%(LUN);
            shared.elementi++;
            /* Controllo se devo sbloccare il lettore */
            if (shared.blocklet != 0)
                pthread_cond_signal(&shared.lettore);
            /* Rilascio la mutua esclusione */
            pthread_mutex_unlock(&shared.mutex);
            /* ... perdo tempo (in modo ORRIBILE!!!!) */
            for(j=0; j<DELAY_WRITER; j++);
        }
    }
    return NULL;
}

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

    local[LUN] = 0;

    for (i=0; i<2*CICLI; i++) {
        for (k=0; k<LUN; k++) {
            /* Come prima cosa acqisisco la mutua esclusione */
```

```
            pthread_mutex_lock(&shared.mutex);
            if (shared.elementi == 0) {
                /* Segnalo che mi sto sbloccando */
                shared.blocklet++;
                pthread_cond_wait(&shared.lettore, &shared.mutex);
                /* Segnalo che mi sono bloccato */
                shared.blocklet--;
            }
            /* Leggo un carattere e aggiorno i vari campi */
            local[k] = shared.scritta[shared.primo];
            shared.primo = (shared.primo+1)%(LUN);
            shared.elementi--;
            /* Controllo se devo sbloccare uno scrittore */
            if (shared.blockscri != 0)
                pthread_cond_signal(&shared.scrittori);
            /* Rilascio la mutua esclusione */
            pthread_mutex_unlock(&shared.mutex);
            /* ... perdo tempo (in modo ORRIBILE!!!!) */
            for(j=0; j<DELAY_READER; j++);
        }
        printf("Stringa = %s \n", local);
    }
    return NULL;
}
```

L'esempio verte sempre sul problema dei lettori e degli scrittori. 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.

Per colmare questo vuoto 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 3.3.1) 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[1].

value – Valore iniziale del semaforo.

Valore restituito :

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.

Valore restituito :

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

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.

Per completezza riportiamo le seguenti primitive che operano sui semafori :

```
#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.

3.3.2 Un esempio

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

```
/* Programma di esempio dei semafori ....
 * Implementazione con il meccanismo semaforico del meccanismo
 * visto nel file condition.c
 */

#include <semaphore.h>
```

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

/* Numero di cicli di lettura/scrittura che vengono fatti dai thread */
#define CICLI 1
/* Lunghezza del buffer */
#define LUN 20
/* Numero di cicli di attesa a vuoto di uno scrittore */
#define DELAY_WRITER 200000
/* Numero di cicli di attesa a vuoto di uno scrittore */
#define DELAY_READER 2000000

/* Memoria Condivisa fra i thread ... */
struct {
    char scritta[LUN+1];
    /* variabili per la gestione del buffer */
    int primo, ultimo;
    /* Variabili semaforiche */
    sem_t mutex, piene, vuote;
} shared;

/* Forward declaration */
void *scrittore1(void *);
void *scrittore2(void *);
void *lettore(void *);

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

    /* Fase di inzializzazione delle strutture dati */
    for(i=0; i<LUN; i++) {
        shared.scritta[i] = 'x';
    }
    shared.primo = shared.ultimo = 0;
    /* Fine della fase di inizializzazione delle strutture dati */

    /* Fase di inizializzazione dei semafori */
    if (sem_init(&shared.mutex, 0, 1) == -1) {
        printf("Non sono riuscito ad inizializzare
```

```
shared.mutex\n");
        return 1;
    }

    if (sem_init(&shared.piene, 0, 0) == -1) {
        printf("Non sono riuscito ad inizializzare
shared.piene\n");
        return 1;
    }

    if (sem_init(&shared.vuote, 0, LUN) == -1) {
        printf("Non sono riuscito ad inizializzare
shared.vuote\n");
        return 1;
    }
    /* Fine della fase di inizializzazione dei semafori */

    /* Ora puo' iniziare la fase di creazione dei thread ... */

    /* Setto il concurrency level a 3 */
    pthread_setconcurrency(3);

    /* A questo punto posso creare i thread .... */
    res = pthread_create(&lTID, NULL, lettore, NULL);
    if (res != 0) {
        printf("Errore nella creazione del primo thread\n");
        return -1;
    }

    res = pthread_create(&s1TID, NULL, scrittore1, NULL);
    if (res != 0) {
        printf("Errore nella creazione del secondo thread\n");
        /* Devo far terminare il thread che ho creato */
        pthread_kill(s1TID, SIGKILL);
        return -1;
    }

    res = pthread_create(&s2TID, NULL, scrittore2, NULL);
    if (res != 0) {
        printf("Errore nella creazione del terzo thread\n");
        pthread_kill(lTID, SIGKILL);
```

```
        pthread_kill(s1TID, SIGKILL);
        return -1;
    }

    /* A questo punto aspetto che i tre thread finiscano ... */
    pthread_join(s1TID, NULL);
    pthread_join(s2TID, NULL);
    pthread_join(lTID, NULL);

    printf("E' finito l'esperimento ....\n");
    return 0;
}

/* Codice del primo thread scrittore */
void *scrittore1(void *in)
{
    int i, j, k;

    for (i=0; i<CICLI; i++) {
        for(k=0; k<LUN; k++) {
            /* Controllo che il buffer non sia pieno */
            sem_wait(&shared.vuote);
            /* Acquisisco la mutua esclusione */
            sem_wait(&shared.mutex);
            /* Inserisco un carattere e aggiorno le strutture dati */
            shared.scritta[shared.ultimo] = '-';
            shared.ultimo = (shared.ultimo+1)%(LUN);
            /* Libero il semaforo di mutua esclusione */
            sem_post(&shared.mutex);
            /* Segnalo che ho aggiunto un carattere */
            sem_post(&shared.piene);
            /* ... perdo un poco di tempo */
            for(j=0; j<DELAY_WRITER; j++);
        }
    }
    return NULL;
}

/* Codice del secondo thread scrittore */
void *scrittore2(void *in)
{
```

```
    int i, j, k;

    for(i=0; i<CICLI; i++) {
        for(k=0; k<LUN; k++) {
            /* Controllo che il buffer non sia pieno */
            sem_wait(&shared.vuote);
            /* Acquisisco la mutua esclusione */
            sem_wait(&shared.mutex);
            /* Inserisco un carattere e aggiorno le strutture dati */
            shared.scritta[shared.ultimo] = '+';
            shared.ultimo = (shared.ultimo+1)%(LUN);
            /* Libero il semaforo di mutua esclusione */
            sem_post(&shared.mutex);
            /* Segnalo che ho aggiunto un carattere */
            sem_post(&shared.piene);
            /* ... perdo un poco di tempo */
            for(j=0; j<DELAY_WRITER; j++);
        }
    }
    return NULL;
}

/* Thread che legge i dati */
void *lettore(void *in)
{
    int i, j, k;
    /* Buffer locale in cui metto i caratteri mano a mano
     * che vengono letti.
     */
    char local[LUN+1];

    local[LUN] = 0;

    for (i=0; i<2*CICLI; i++) {
        for(k=0; k<LUN; k++) {
            /* Controllo che il buffer non sia vuoto */
            sem_wait(&shared.piene);
            /* Acquisisco la mutua esclusione */
            sem_wait(&shared.mutex);
            /* Aggiungo un carattere e aggiorno le strutture dati */
            local[k] = shared.scritta[shared.primo];
```

```
            shared.primo = (shared.primo+1)%(LUN);
            /* Libero il semaforo di mutua esclusione */
            sem_post(&shared.mutex);
            /* Segnalo che ho letto un carattere */
            sem_post(&shared.vuote);
            /* ... perdo un poco di tempo */
            for(j=0; j<DELAY_READER; j++);
        }
        printf("Stringa = %s \n", local);
    }
    return NULL;
}
```

L'esempio è essenzialmente analogo all'esempio , 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**.

3.3.3 Esempio di architettura Client Server

Il paradigma client-server si basa sull'assunto che esiste un certo insieme di entità che ha bisogno di ottenere un certo servizio da un'altra entità e che esiste un meccanismo di comunicazione a scambio di messaggi. A ogni entità che richiede il servizio viene dato il nome di **client**, mentre a quella che lo offre il nome di **server**.

Con il termine entità si intende un'ampia casistica che può andare da un host in rete fino a un thread. Nel seguito intenderemo in modo implicito che client e server siano dei task o dei thread.

Questa semplice spiegazione nasconde in realtà la complessità del problema. Una complicazione che sorge subito è che l'architettura così prospettata si presta subito ad un attacco, che in gergo tecnico prende il nome di *denial of service*.

Tale attacco viene portato da un client che vuole negare, per scopi più o meno leciti, l'uso del server a tutti gli altri client. Quello che il client deve fare è richiedere al server un servizio valido ma il cui tempo di elaborazione sia così lungo da bloccare, di fatto, il server e quindi renderlo inaccessibile a tutti gli altri client.

Per evitare tale tipo di attacco il server agisce nel seguente modo :

- Si mette in attesa sul canale di comunicazione.

- All'atto della ricezione di una richiesta si servizio genera un task o un thread.
- Tale task o thread apre un canale di comunicazione con il client e porta a termine la richiesta.

`Esercizio:` Si dica perchè, con l'architettura appena discussa, non è possibile un attacco di tipo **denial of service**.

3.3.4 Architettura Client-Server

In questo listato è esemplificata un'architettura client-server. Il server è naturalmente molto semplice e si limita ad invertire una stringa che gli viene passata tramite la memoria condivisa.

```
/* Programma di esempio di un client e di un server
 * sfruttando i thread e i semafori privati.
 */
#include <stdio.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>

/* Lunghezza massima del buffer */
#define LUN 40
/* Numero di client che viene generato */
#define NUM_CLIENT 30

struct{
    /* buffer che conterra' la stringa */
    char buffer[LUN];
    /* semaforo di mutua esclusione */
    sem_t mutex;
    /* Semaforo privato su cui si blocca il client */
    sem_t *private;
    /* Puntatore alla zona in cui verra' messo il risultato */
    char *risultato;
}shared;

sem_t richiesta;
```

```
/* Forward declaration */
void *client(void *);
void *server(void *);
void *realServer(void *);

int main(void)
{
    pthread_t clientID[NUM_CLIENT], serverID;
    int res, i, j;

    /* Fase di inzializzazione delle strutture dati */
    /* Ogni stringa C deve terminare con 0!!!!! */
    shared.buffer[0] = 0;

    /* Fine della fase di inizializzazione delle strutture dati */

    /* Fase di inizializzazione dei semafori */
    if (sem_init(&shared.mutex, 0, 1) == -1) {
        printf("Non sono riuscito ad inizializzare
shared.mutex\n");
        return 1;
    }

    if (sem_init(&richiesta, 0, 0) == -1) {
        printf("Non sono riuscito ad inizializzare
shared.mutex\n");
        return 1;
    }

    /* Ora inizia la fase di creazione dei thread */
    pthread_setconcurrency(1+2*NUM_CLIENT);

    /* Come prima cosa creo il server */
    res = pthread_create(&serverID, NULL, server, NULL);

    if (res != 0) {
        printf("Non ho potuto creare il server \n");
        exit(-1);
    }

    for(i=0; i<NUM_CLIENT; i++) {
```

```
        res = pthread_create(&clientID[i], NULL, client, NULL);
        if (res != 0) {
            printf("Non ho potuto creare i client \n");
            for(j=0; j<i; j++) {
                pthread_kill(clientID[j], SIGKILL);
            }
            pthread_kill(serverID, SIGKILL);
            exit(-1);
        }
    }

    /* A questo punto aspetto i thread dei client */
    for(i=0; i<NUM_CLIENT; i++)
        pthread_join(clientID[i], NULL);

    pthread_kill(serverID, SIGKILL);

    return 0;
}

/* Codice del server che genera un nuovo thread ogni volta
 * che un nuovo client fa una richiesta. */
void *server(void *in)
{
    pthread_t tmp;

    for(;;) {
        sem_wait(&richiesta);
        tmp = pthread_create(&tmp, NULL, realServer, NULL);
        if (tmp != 0) {
            printf("Non sono riuscito a creare un server ....\n");
            printf("Se continuo sono in deadlock!!!!!\n");
            exit(-1);
        }
        pthread_detach(tmp);
    }
}

/* Server vero che inverte la stringa */
void *realServer(void *in)
{
```

```
    char buf[LUN], *risu;
    sem_t *priv;
    int i, j, len;

    /* Come prima cosa copio i dati nello spazio locale */
    priv = shared.private;
    strcpy(buf, shared.buffer);
    risu = shared.risultato;
    len = strlen(buf);
    sem_post(&shared.mutex);

    /* Ora svolgo il lavoro del server */
    for (i=len-1; i≥0; i--) {
        risu[len-1-i] = buf[i];
    }
    risu[len] = 0;

    /* A questo punto perdo un poco di tempo */
    for(j=0; j<20000000; j++);

    sem_post(priv);
    return NULL;
}

/* Codice dei vari client */
void *client(void *in)
{
    char risu[LUN];
    sem_t priv;

    risu[0] = 0;
    if (sem_init(&priv, 0, 0) == -1) {
        printf("Non sono riuscito ad inizializzare un semaforo
privato\n");
        pthread_exit(NULL);
    }

    sem_wait(&shared.mutex);
    shared.risultato = risu;
    sprintf(shared.buffer, "thread numero %d", (int)pthread_self());
    shared.private = &priv;
```

```
    sem_post(&richiesta);
    sem_wait(&priv);
    printf("Thread %d -- Risultato = %s\n", (int)pthread_self(), risu);
    return NULL;
}
```

Nell'esempio viene introdotta anche la seguente funzione :

```
#include <pthread.h>
int pthread_detach(pthread_t *thread);
```

Argomenti in ingresso :

thread – Identificatore del thread.

Valore restituito :

0 – In caso di successo

$\neq 0$ – In caso di insuccesso.

Normalmente il kernel mantiene un certa quantità di informazioni per ogni thread. Tali informazioni servono, dopo la terminazione del thread, per permettere ad altri thread di effettuare una `pthread_join()` su di esso; ciò comporta che nessuno effettua una join su un thread si può verificare uno spreco di memoria. La `pthread_detach()` serve proprio a soecificare che nessuno effettuerà una join sul thread specificato, e che quindi le informazioni ad esso relative possono essere cancellate direttamente alla sua terminazione.