Programmazione su sistemi UNIX

Sistemi Operativi

Mario Di Raimondo

C.d.L. in Informatica (laurea triennale)
Dipartimento di Matematica e Informatica – Catania

A.A. 2011-2012

Chiamate di sistema

In questa parte ci occuperemo di alcuni aspetti fondamentali inerenti il problema della progettazione e realizzazione di programmi nei sistemi UNIX (in particolare su Linux).
In particolare, ci si occuperà dell'interfaccia tra il programmatore ed il sistema operativo.
In genere i programmi, per svolgere il loro compito, hanno bisogno dell'esecuzione di alcune mansioni ausiliarie strettamente legate alla gestione delle risorse del sistema.
Il meccanismo che permette ad un'applicazione di invocare una mansione ausiliaria è il meccanismo delle system calls (chiamate di sistema).

Tipo di chiamate di sistema

Nei sistemi UNIX le chiamate di sistema sono in genere suddivise in categorie a secondo del tipo di risorse da gestire:

- gestione dei file;
- gestione di processi;
- comunicazione e sincronizzazione tra processi.

Qui vedremo solo alcune (una trattazione esaustiva sarebbe fuori dagli scopi del corso). Il linguaggio di riferimento sarà il Linguaggio C, le interfacce di sistema lo utilizzano come riferimento poiché il kernel stesso dei sistemi UNIX in genere (ed è il caso di Linux) è scritto in C (con l'ausilio di codice scritto in assembly).

Serve aiuto?

Così come avete visto nella parte dedicata alla shell, anche qui le man pages dei sistemi UNIX ci vengono in aiuto. Sono disponibili delle pagine di manuale per ognuno delle principali chiamate di sistema.
Il modo in cui richiamare la pagina è il solito: `man chiamata`
A volte, ci sono chiamate di sistema che sono omonime a comandi. Le man page sono organizzate in sezioni. Alcuni esempi di sezioni:

- 1: programmi eseguibili e comandi di shell;
- 2: chiamate al sistema (funzioni fornite dal kernel);
- 3: chiamate alle librerie (funzioni all'interno delle librerie di sistema).

In caso di più pagine con lo stesso nome, si può specificare la sezione usando la sintassi: `man numero_sezione nome_pagina`
Esempio: `man 2 chown`

Descrittori di file (1)

Ogni processo può effettuare operazione di lettura/scrittura su uno o più file. Ogni operazione fa riferimento ad un determinato file.
Il kernel associa ad ogni processo una tabella dei file aperti dal processo stesso. All'atto della creazione di un file, o se è già esistente e si apre, il sistema associa ad esso un intero non negativo, denominato descrittore di file, che identifica il canale di input/output.
Esso rappresenta un collegamento tra il processo ed il file ad esso associato. Non è altro che l'indice nella tabella dei file aperti dal processo.
All'atto della creazione di un processo, le prime tre allocazioni di tale tabella sono riservate nel seguente modo:

- 0: standard input
- 1: standard output
- 2: standard error

Descrittori di file (2)

Ogni elemento della tabella dei file aperti ha un puntatore di lettura/scrittura chiamato file pointer. Tale puntatore è inizialmente (in apertura o creazione) pari a 0, per indicare che punta l'inizio del file. Ad ogni operazione di lettura/scrittura, il puntatore viene spostato di altrettanti byte. Indica il punto in cui avverrà la prossima operazione di lettura/scrittura.
E' possibile spostare tale puntatore in una posizione arbitraria attraverso apposite chiamate di sistema.
Il numero di file che ogni processo può tenere aperti contemporaneamente è limitato (che dipende dalla dimensione della tabella dei file). E' quindi buona regola mantenere a minimo il numero di file aperti e chiudere ogni file a cui non si ha l'intenzione di accedere nell'immediato futuro.

Creazione di un file

Per creare un nuovo file in un sistema UNIX viene utilizzata la chiamata di sistema `creat()`. La sintassi è la seguente:

```
int creat(char *file_name, int mode)
```

- `file_name`: puntatore alla stringa di caratteri che contiene il nome del file da creare (pathname assoluto o relativo);
- `mode`: specifica i permessi di accesso al file da creare.

Se il file esiste già, l'effetto della chiamata è quello di azzerare il contenuto dello stesso (portando la dimensione a zero) lasciando inalterati i diritti di accesso precedenti (quelli specificati con `mode` vengono ignorati).
La chiamata a `creat()` ritorna un valore per determinare l'esito dell'operazione. Se il valore è non negativo, l'operazione di creazione è andata a buon fine ed il valore riportato è esattamente il suo `descrittore di file`. Tale numero può essere usato come riferimento per le successive operazioni sul file da parte del processo.

Creazione di un file: esempio

creat.c

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

int main() {
   int fd;
   if ( (fd=creat("pippo.txt",0600)) == -1) {
      printf("Errore nella chiamata creat \n");
      exit(1);
   }
   printf("File creato con successo con descrittore di file %d \n",fd);
}
```

Circa i permessi di accesso

Le chiamate di sistema che fanno uso di un parametro che specifica i diritti di accesso per i file possono fare uso di alcune costanti predefinite che possono essere combinati con l'operatore |:

- `S_IRUSR`: permesso di lettura, proprietario;
- `S_IWUSR`: permesso di scrittura, proprietario;
- `S_IXUSR`: permesso di esecuzione, proprietario;
- `S_IRGRP`: permesso di lettura, gruppo;
- `S_IWGRP`: permesso di scrittura, gruppo;
- `S_IXGRP`: permesso di esecuzione, gruppo;
- `S_IROTH`: permesso di lettura, altri;
- `S_IWOTH`: permesso di scrittura, altri;
- `S_IXOTH`: permesso di esecuzione, altri.

Tali costanti si trovano nel file `sys/stat.h` che si deve quindi includere.

Aprire un file (1)

Nel caso in cui il file da gestire esiste già, per fare delle operazioni su di esso è necessario comunque associare un descrittore di file ad esso (nell'ambito del processo corrente).
Per fare ciò viene usata la chiamata di sistema `open()` che ha la seguente sintassi:

```
int open(char *file_name, int option_flags[, int mode])
```

- `file_name`: puntatore alla stringa di caratteri che contiene il nome del file da aprire;
- `option_flags`: specifica la modalità di apertura;
- `mode`: specifica dei permessi nel caso in cui sia necessario creare il file.

Aprire un file (2)

La modalità di apertura `option_flags` è un intero i cui bit rappresentano alcune opzioni da usare nell'apertura del file. I bit desiderati sono attivati utilizzando apposite costanti definite nell'header `/usr/include/fcntl.h` combinandoli con l'operatore | (OR bit-a-bit):

- `O_RDONLY`: apre il file in sola lettura;
- `O_WRONLY`: apre il file in sola scrittura;
- `O_RDWR`: apre il file in lettura e scrittura (combinazione delle due precedenti);
- `O_APPEND`: apre il file e posiziona il puntatore di posizione alla fine, cosicché una eventuale operazione di scrittura accodi i blocchi scritti a quelli già esistenti;
- `O_CREAT`: se il file non esiste, il file viene creato con i diritti di accesso specificati da `mode`;
- `O_TRUNC`: se il file esiste lo tronca a dimensione zero.

In effetti si deve scegliere uno tra solo-lettura, solo-scrittura e lettura-scrittura e poi eventualmente combinare le altre modalità con l'OR.

Aprire un file (3)

La funzione `open()` apre un file esistente e riporta un descrittore di file per esso relativamente al processo chiamante, dopo aver creato una apposita voce nella tabella dei file aperti. Se il file non esiste ed è stata specifica l'opzione `O_CREAT`, allora il file viene creato con gli eventuali permessi specificati con il parametro opzionale `mode`.
Se l'operazione di apertura/creazione fallisce, la funzione riporta il valore `-1`.
Una chiamata del tipo `open(filename, O_RDWR|O_TRUNC|O_CREAT, 0660)` è del tutto equivalente a `creat(filename, 0660)`, infatti se il file esiste già lo apre e lo tronca portando il puntatore all'inizio del file, il tutto mantenendo i diritti precedenti. Se il file non esiste, entrambe le chiamate creano il file in lettura-scrittura con gli stessi diritti di accesso.

Chiudere un file aperto

Per chiudere un file che si è precedentemente creato/aperto e che non si pensa più di usare nel breve tempo, si può usare la chiamata `close()` per liberare la relativa voce nella tabella dei file aperti ed il relativo descrittore di file.
La sintassi è la seguente:
`int close(int ds_file)`

- `ds_file`: il descrittore del file da chiudere.

Esempio di apertura e chiusura

openclose.c

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

int main() {
   int fd;
   char *name="pippo.txt";

   fd=open(name,O_RDWR);
   if ( fd == -1) {
      printf("Errore nell'apertura del file %s. \n",name);
      exit(1);
   }
   printf("Il file %s e' stato aperto con descrittore di file %d. \n",name,fd);
   close(fd);
}
```

Lettura da un file

Per leggere dei dati da un file precedentemente aperto si può utilizzare la chiamata di sistema `read()`. La sintassi completa è:

```
int read(int ds_file, char *buffer_pointer,
unsigned transfer_size)
```

- `ds_file`: un descrittore di un file valido da cui leggere i dati;
- `buffer_pointer`: un puntatore ad una area di memoria temporanea su cui copiare i dati letti dal file (un buffer);
- `transfer_size`: specifica il numero di byte (caratteri) che si desidera leggere dal file.

In caso di successo, il numero di byte effettivamente letti viene riportato come valore di ritorno (quindi positivo).

Esempio di lettura da file

read.c

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

int main() {
   int fd,n;
   char *name="pippo.txt";
   char buffer[1024];

   if ( (fd=open(name,O_RDONLY)) == -1 ) {
      printf("Errore nell'apertura del file %s. \n",name);
      exit(1);
   }
   n=read(fd,buffer,10);
   if ( n==-1 ) {
      printf("Errore durante la lettura. \n");
      exit(1);
   }
   buffer[n]='\0';
   printf("ho letto: %s \n",buffer);

   close(fd);
}
```

Esempio: conteggia la dimensione di un file

count.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#define BUFSIZE 1024
int main(int argc, char *argv[]) {
   int fd,size;
   int total=0;
   char buffer[BUFSIZE];
   if (argc != 2) { printf("utilizzo: %s file\n",argv[0]); exit(1);  }
   // apre il file sorgente in sola lettura
   fd=open(argv[1],O_RDONLY);
   if (fd == -1) {
      perror(argv[1]);
      exit(1);
   }
   // copia tutti i dati in memoria per conteggiare la dimensione
   do {
      size=read(fd,buffer,BUFSIZE);
      if (size == -1 ) { perror(argv[1]); exit(1); }
      total += size;
      printf("DEBUG: ho letto %d byte\n",size);
   } while (size > 0);
   printf("La dimensione totale e' di %d byte\n",total);
   close(fd);
}
```

Scrittura su un file

Per scrivere dei dati su un file usiamo la chiamata di sistema `write()`. La sintassi completa è:

```
int write(int ds_file, char *buffer_pointer,
unsigned transfer_size)
```

- `ds_file`: un descrittore di un file valido su cui si vuole scrivere i dati;
- `buffer_pointer`: un puntatore ad una area di memoria che contiene i dati da scrivere sul file;
- `transfer_size`: specifica il numero di byte (caratteri) che si desidera scrivere sul file, a partire dall'inizio del buffer.

Se la dimensione del buffer indirizzato da `buffer_pointer` è inferiore al numero di byte da trasferire, l'effetto sul file sarebbe imprevedibile (scrivendo byte presi a caso dalla memoria) o, nel caso estremo, si potrebbero andare a leggere in un'area di memoria non autorizzata.

Esempio di lettura e scrittura combinati (1)

copy.c

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

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
   int sd,dd,size,result;
   char buffer[BUFSIZE];
   // controlla il numero di parametri
   if (argc != 3) {
      printf("utilizzo: %s sorgente destinazione\n",argv[0]);
      exit(1);
   }
   // apre il file sorgente in sola lettura
   sd=open(argv[1],O_RDONLY);
   if (sd == -1) {
      perror(argv[1]);
      exit(1);
   }
   // apre il file destinazione in sola scrittura, con troncamento e creazione
   dd=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0660);
   if (dd == -1) {
      perror(argv[2]);
      exit(1);
   }
```

Esempio di lettura e scrittura combinati (2)

copy.c

```
   // copia i dati dalla sorgente alla destinazione
   do {
      // legge fino ad un massimo di BUFSIZE byte dalla sorgente
      size=read(sd,buffer,BUFSIZE);
      if (size == -1 ) {
         perror(argv[1]);
         exit(1);
      }
      // scrive i byte letti
      result=write(dd,buffer,size);
      if (result == -1) {
         perror(argv[2]);
         exit(1);
      }
   } while (size > 0);
   // chiude i file prima di uscire
   close(sd);
   close(dd);
}
```

Spostarsi all'interno di un file

Abbiamo detto che per ogni file aperto viene mantenuto un indice che rappresenta la posizione corrente (offset).
Per fare degli spostamenti si usa la chiamata `lseek()`:
`long lseek(int ds_file, long offset, int option)`

- `ds_file`: il descrittore del file;
- `offset`: il numero di byte di cui ci si vuole spostare;
- `option`: il tipo di spostamento che si vuole fare:
  - `SEEK_SET (0)`: da inizio file
  - `SEEK_CUR (1)`: dalla posizione corrente
  - `SEEK_END (2)`: dalla fine del file

Riporta la nuova posizione acquisita o `-1` in caso di errore (nel cui caso la posizione resta invariata).
Una invocazione del tipo `pos=lseek(fd,0L,SEEK_CUR)` può essere usata per recuperare la posizione corrente.

Duplicare il descrittore di un file

E' possibile duplicare il descrittore di un file nella tabella dei file aperti, in modo tale che:

- il nuovo descrittore si riferisce allo stesso file, eredita il puntatore di lettura/scrittura e la modalità di accesso;
- il nuovo descrittore utilizzerà l'entry libera della tabella con indice più piccolo.

La chiamata di sistema è `dup()`:

```
int dup(int file_descriptor)
```

Dove `file_descriptor` è il descrittore di file da duplicare. La funzione riporta il numero del nuovo descrittore creato o `-1` in caso di problemi.
La duplicazione può essere usata per fare la redirezione dei canali di input-output.

Redirezione dello standard input

In questo esempio lo standard input viene preso dal file passato come parametro e poi viene eseguito il comando `more` (che eredita lo standard input).

mymore.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define STDIN 0

int main(int argc, char *argv[]) {
   int fd;
   if (argc == 1) exit(1);
   fd = open(argv[1],O_RDONLY);  // apro il file in lettura
   close(STDIN);                 // chiudo lo standard input
   dup(fd);                   // duplico il descrittore del file
   close(fd);                    // chiude il vecchio descrittore (ne resta una copia)
   execlp("more","more",NULL);      // eseguo "more" (vedremo piu' avanti i dettagli)
}
```

Lavorare con gli stream

Le funzionalità che abbiamo appena visto per operare sui file del filesystem sono operazioni a basso livello (low-level), operano in modo molto simili a ciò che fa il kernel. Vengono spostati blocchi grezzi di dati senza curarsi di cosa ci sta dentro.
Un modo alternativo di operare sui file è quello di utilizzare gli stream: permettono di utilizzare funzioni più ad alto livello che semplificano, a volte, il lavoro del programmatore quando si trattano file di testo.
Ogni stream identifica un file (o più in generale un canale di I/O) e rappresenta una astrazione a più alto livello rispetto ai file descriptor: in ogni caso, quando viene aperto uno stream per un file, viene creato un relativo file descriptor nella tabella dei file aperti.
Per operare sugli stream in genere è sufficiente includere gli header `stdio.h`.
Alla creazione del processo vengono creati tre flussi standard: `stdin`, `stdout` e `stderr`.

Apertura di uno stream (1)

Per ogni funzione a basso livello vista in precedenza, esistono delle controparti che lavorano sugli stream. Poi ci sono anche altre funzionalità più avanzate che giustificano l'esistenza stessa degli stream.
Per aprire uno stream relativamente ad un file esistente:

```
FILE *fopen(char *path, char *mode)
```

Apre un file con pathname riferito da `path` e con la modalità di apertura specificata nella stringa riferita da `mode`:

- `r` o `r+`: il file viene aperto in sola lettura (o lettura/scrittura) e lo stream è posizionato all'inizio del file;
- `w` o `w+`: se il file esiste questo viene aperto in sola scrittura (o lettura/scrittura), viene troncato e il flusso si posiziona all'inizio; se non esiste viene creato;
- `a` o `a+`: se il file non esiste allora viene creato, se esiste viene aperto in scrittura (o lettura/scrittura) e lo stream viene posizionato alla fine.

Se la chiamata non va a buon fine, viene riportato un puntatore a `NULL`.

Apertura di uno stream (2)

Se l'apertura/creazione va a buon file, viene riportato un puntatore ad una struttura standard chiamata `FILE` (in realtà `FILE` è un alias di una struttura). Tale puntatore sarà il riferimento allo stream per tutto il resto dell'esecuzione del processo.

E' possibile aprire uno stream collegato ad un file descriptor già creato con la variante:
`FILE *fdopen(int fd, char *mode)`
Il file descriptor `fd` non viene duplicato (tipo con `dup`) ed i modi di apertura specificati in `mode` devono essere compatibili con quelli già specificati nella tabella dei file aperti.

Una stream si può chiudere con la chiamata `int fclose(FILE *fp)`.

Lettura e scrittura di blocchi su uno stream

Esistono funzioni molto simili alle chiamate `read()` e `write()` che lavorano per blocchi:

```
int fread(void *ptr, int size, int num, FILE *stream)
int fwrite(void *ptr, int size, int num, FILE *stream)
```

Queste funzioni leggono/scrivono `num` blocchi di dimensione `size` utilizzando il buffer riferito da `ptr` per i blocchi letti/scritti.
Le due funzioni riportano il numero di elementi (NON di byte!) letti o scritti. Se avviene un errore (o viene raggiunta la fine del file) un numero inferiore a `num` viene riportato (non si fa distinzione tra l'errore e l'end-of-file).
Per vedere se si è raggiunto la fine del file si può usare la funzione `int feof(FILE *stream)`.

Leggere e scrivere caratteri singoli

Da uno stream è possible leggere e scrivere caratteri singoli (byte) utilizzando le seguenti funzioni:

```
int fgetc(FILE *stream)
int fputc(int c, FILE *stream)
```

`fgetc()` legge un carattere dallo stream `stream` e lo ritorna come valore. Se viene raggiunta la fine del file, viene riportato il valore speciale `EOF` (che in effetti corrisponde a `-1`).
In effetti viene riportato un `int`, ma è il risultato di un cast da un `unsigned char`.

La chiamata `fputc` scrive il carattere contenuto in `c` sullo stream `stream` (subito dopo averne fatto un cast ad `unsigned char`).
Ritorna il valore appena scritto o `EOF` in caso di errore.

Esempio: copia di file carattere per carattere con stream

copystream.c

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

int main(int argc, char *argv[]) {
   FILE *in,*out;
   int c;
   if (argc != 3) { printf("utilizzo: %s sorgente destinazione\n",argv[0]); exit(1); }
   // apre lo stream sorgente in sola lettura
   if ((in=fopen(argv[1],"r")) == NULL) {
      perror(argv[1]);
      exit(1);
   }
   // apre/crea lo stream destinazione in sola scrittura (con troncamento)
   if ((out=fopen(argv[2],"w")) == NULL) {
      perror(argv[2]);
      exit(1);
   }
   // copia i dati dalla sorgente alla destinazione carattere per carattere
   while ( (c = fgetc(in)) != EOF )
      fputc(c,out);
   // chiude gli stream
   fclose(in);
   fclose(out);
   exit(0);
}
```

Leggere e scrivere stringhe

Per leggere una stringa da uno stream si può usare:
`char *fgets(char *s, int size, FILE *stream)`
Legge dallo stream `stream` un numero di caratteri pari al più a `(size-1)` e li memorizza nel buffer puntato da `s`. La lettura termina nel caso di incontri la fine del file o un carattere di ritorno a capo sul file (`'\n'`). Nel buffer, subito dopo la fine della stringa appena letta, viene inserito un carattere `'\0'` per rendere il buffer `s` un stringa valida in C.
La chiamata `fgets()` riporta `s` stesso in caso di successo o `NULL` in caso di errore o fine file.

Per scrivere si può utilizzare la chiamata:
`int fputs(char *s, FILE *stream)`
Viene scritta la stringa puntata da `s` (senza il carattere `'\0'` finale).

Esempio: reimplementazione di cat usando gli stream

cat.c

```
// legge un file di testo (o lo standard input) e lo visualizza
#include <stdio.h>
#include <stdlib.h>

#define BUFSIZE 2048

int main(int argc, char *argv[]) {
   FILE *in;
   char buffer[BUFSIZE];
   int c;
   if (argc >= 2) {  // e' stato specificato almeno un parametro
      if ((in=fopen(argv[1],"r")) == NULL) { perror(argv[1]); exit(1); }
   } else      // legge dallo standard input
      in=stdin;
   // copia i dati dalla sorgente alla destinazione carattere per carattere
   while ( (fgets(buffer,BUFSIZE,in)) != NULL )
      printf("%s",buffer);
   // chiude gli stream
   fclose(in);
   exit(0);
}
```

Scrittura avanzata di stringhe

La funzione `printf()` che in genere viene usata per scrivere stringhe formattate con valori aggiuntivi a video può in effetti essere utilizzata (o meglio, una sua variante) per scrivere su un generico stream. La variante è la seguente:

```
int fprintf(FILE *stream, char *format, ...)
```

In effetti una chiamata del tipo `printf(...)`, corrisponde ad una invocazione del tipo `fprintf(stdout,...)`.

La sintassi di formattazione è identica a quella che conoscete per `printf()`. Alcuni esempi di richiamo: `%d` per gli interi, `%o` e `%x` per gli interi in ottale ed esadecimale, `%c` per un carattere, `%s` per una stringa, `%f` per un float, ecc.

Il valore ritornato da `fprintf()` è il numero di caratteri scritti.

Lettura avanzata di stringhe

La funzione `scanf()` può essere utilizzata per leggere dallo standard input (`stdin`) una stringa formattata. Esiste l'analogo `fscanf()` che compie lo stesso compito ma leggendo da uno stream specifico. In effetti, `scanf()` utilizza implicitamente lo stream predefinito `stdin`.

```
int fscanf(FILE *stream, char *format, ...)
```

Gli stessi simboli speciali `%?` visti per `fprintf()` si possono utilizzare per `fscanf()`; per ognuno di essi deve essere specificato un puntatore alla variabile che conterrà il valore letto.
La chiamata riporta il numero di valori che sono stati assegnati (che potrebbero anche meno di quelli specificati per mancanza di input o per differenze nel parsing).
L'utilizzo di `fscanf()` non è sempre semplice e spesso può portare ad comportamenti imprevisti se il file non è formattato esattamente.

Altre funzioni sugli stream

Ecco qualche altra funzione sugli stream:

- `int fseek(FILE *stream, long offset, int mode)`
  sposta la posizione di `offset` byte, utilizzando il riferimento specificato da `mode` (che ha lo stesso utilizzo che in `lseek()`);
- `long ftell(FILE *stream)`
  riporta la posizione corrente nello stream;
- `void rewind(FILE *stream)`
  "riavvolge lo stream", portando la posizione all'inizio dello stream;
- `int fflush(FILE *stream)`
  forza il sistema a scrivere tutti i "dati in sospeso" contenuti nei buffer (a livello di utente).

Creare un link

Nella prima parte del corso abbiamo già parlato degli hard-link e dei soft-link. Vediamo come creare un hard-link di un file attraverso la chiamata di sistema `link()`:

```
int link(char *pathname, char *alias)
```

- `pathname`: puntatore alla stringa contenente il pathname del file esistente di cui creare l'hard-link;
- `alias`: puntatore al nome dell'hard-link da creare.

La funzione riporta `0` in caso di successo o `-1` se l'operazione è fallita (ad esempio non si hanno i diritti necessari o si è cercato di fare un hard-link a cavallo di due file-system diversi o che non supportano la primitiva stessa).

Esiste anche la chiamata di sistema per creare i link simbolici:

```
int symlink(char *pathname, char *alias)
```

Queste chiamate stanno in `unistd.h`.

Rimuovere un link... ovvero cancellare

L'operazione inversa compiuta dalla chiamata `link()` si effettua con `unlink()`: rimuove il riferimento ad un file esistente. In pratica viene decrementato il contatore dei link all'interno dell'inode e se esso arriva a `0`, allora viene cancellato anche l'inode ed il contenuto del file. Se c'erano più di un link, il file continua ad esistere ed rimane referenziato dai rimanenti link.
In effetti, le cancellazioni sotto UNIX si fanno con la chiamata `unlink()`:
`int unlink(char *pathname)`

- `pathname`: puntatore alla stringa contenente il pathname del riferimento da cancellare.

Ritorna `0` in caso di successo, `-1` in caso di errore.

Rinomina/spostamento tramite link

move.c

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

int main(int argc, char *argv[]) {
   if (argc != 3) {
      fprintf(stderr,"utilizzo: %s vecchionome nuovonome\n",argv[0]);
      exit(1);
   }
   if (link(argv[1],argv[2]) == -1) {
      perror(argv[2]);
      exit(1);
   }
   if (unlink(argv[1]) == -1) {
      perror(argv[1]);
      exit(1);
   }
   exit(0);
}
```

Altre funzioni di servizio

Alcune chiamate di sistema per la gestione e manipolazione del filesystem:

- `int chmod(char *path, mode_t mode)`
  cambia i diritti di accesso dell'oggetto specificato da `path`, impostando i diritti a `mode`; quest'ultimo utilizza le stesse maschere di bit che abbiamo visto in precedenza per `creat()`;
- `int chown(char *path, uid_t owner, gid_t group)`
  cambia il proprietario nell'utente specificato da `owner` ed il gruppo `group`;
- `int mkdir(char *pathname, mode_t mode)`
  crea un directory specificata da `pathname` con i permessi in `mode`;
- `int rmdir(char *pathname)`
  cancella la directory (vuota) specificata da `pathname`;
- `int chdir(char *pathname)`
  cambia la directory corrente in quella specificata da `pathname`;
- `char *getcwd(char *buf, size_t size)`
  scrive nel buffer `buf` (di dimensione `size`) la directory corrente.

Cos'è un processo?

Un processo può essere considerato in prima approssimazione come un programma in esecuzione. Dato un programma in esecuzione si individuano:

- codice: l'insieme delle istruzioni che lo compongono;
- spazio di memoria: l'attuale stato della memoria occupata dai dati del programma;
- stato del processore: l'attuale valore dei registri della CPU.

Data una terna (codice,memoria,stato processore) abbiamo univocamente definito un programma in esecuzione. Questa terna è detta processo, che quindi è un concetto legato a quello di programma, ma che ne rappresenta una estensione.
Quindi se ad un certo istante abbiamo in esecuzione un certo processo A e sostituiamo i valori attuali con quelli di una terna associata ad un secondo processo B, allora il flusso di esecuzione di quest'ultimo riprenderà dal punto in cui era stato lasciato. Questo è un interlacciamento tra processi e costituisce il concetto di base dei moderni sistemi multi-tasking.

Il kernel e i processi

Il kernel gestisce i processi in esecuzione attraverso una tabella dei processi attivi ed associa ad ogni processo un identificatore numerico chiamato PID che lo individua in modo univoco.
Il PID di un processo è assegnato dal kernel al processo nella fase di creazione del processo stesso.

La suddetta tabella contiene per ogni processo una struttura nota come Process Control Block (PCB), contenente informazioni relative al processo come: il PID del processo, il suo stato (pronto per l'esecuzione, in esecuzione, in attesa, ecc.).

Ogni programma può conoscere il suo PID utilizzando la chiamata di sistema `pid_t getpid()`.

Esempio di recupero del PID attuale

pid.c

```
// visualizza il PID del processo corrente
#include <stdio.h>
#include <stdlib.h>

int main() {
   pid_t pid;
   pid = getpid();
   printf("PID che il kernel ha associato a questo processo: %d\n",pid);
   exit(0);
}
```

Creazione di un processo (1)

Nei sistemi UNIX esiste un unico modo per creare un nuovo processo: utilizzando la chiamata di sistema `pid_t fork()`.
L'effetto di tale chiamata è quello di creare un esatto duplicato del processo corrente, quindi: stesso codice, stessa memoria (nel senso che ne crea una copia esatta) e medesimo stato del processore. Ovviamente avrà un PID diverso: non possono esistere due processi con PID uguale.
Il primo processo sarà detto processo padre, quello generato (il duplicato) sarà detto processo figlio.

Da quanto detto fino adesso si può capire perché il processo `init` è il padre di tutti i processi, infatti ha sempre PID pari a `1`.

Il nome della chiamata sta a rappresentare appunto tale processo di biforcazione: ma dov'è che avviene la biforcazione? Il processo padre ed il processo figlio hanno il medesimo codice ed il medesimo stato: continueranno a compiere l'identico lavoro (in parallelo).

Creazione di un processo (2)

L'unica differenza nel flusso di esecuzione dei due processi imparentati è costituito dal valore di ritorno della chiamata `fork()`:

- al processo padre riporta il `PID` del processo figlio appena creato;
- al figlio restituisce sempre `0`;
- in caso di errore ritorna `-1`.

Quindi i due processi, analizzando il valore di ritorno della chiamata, possono capire se si tratta del padre o del figlio ed agire di conseguenza.

Da notare che in effetti il processo figlio eredita tutto lo stato del processo padre, compreso la tabella dei file aperti, che contiene anche i descrittori dello standard input e standard output. Il padre ed il figlio li condivideranno.

Il processo figlio (ma in generale, ogni processo) può conoscere il PID del suo processo padre (ognuno ha un padre... eccetto `init`) attraverso la chiamata `pid_t getppid()`.

Utilizzo di fork

fork.c

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

int main() {
   pid_t pid;
   printf("PID del processo iniziale: %d\n",getpid());

   pid = fork();

   if (pid == 0)
      printf("Sono il processo FIGLIO [%d] e mio padre ha PID %d.\n",getpid(),getppid());
   else
      printf("Sono il processo PADRE [%d] e mio figlio ha PID %d.\n",getpid(),pid);

   exit(0);
}
```

Altro esempio con fork

fork2.c

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

int main() {
   pid_t pid;
   int i;

   pid = fork();

   if (pid == 0) {
      // processo figlio
      for (i=0; i<200; i++)
         printf("Sono il FIGLIO (%d).\n",i);
      exit(0);
   }

   // processo padre
   for (i=0; i<200; i++)
      printf("Sono il PADRE (%d).\n",i);
   exit(0);
}
```

Esempio di creazione di più figli

multifork.c

```
// esempio di creazione di piu' figli
#include <stdio.h>
#include <stdlib.h>
int main() {
   pid_t pid;
   int i;
   pid = fork();  // creo un primo figlio
   if (pid == 0) {
      // processo figlio 1
      for (i=0; i<20; i++) {
         sleep(rand()%2);
         printf("Sono il FIGLIO 1.\n"); }
      exit(0);
   }
   pid = fork();  // creo un secondo figlio
   if (pid == 0) {
      // processo figlio 2
      for (i=0; i<20; i++) {
         sleep(rand()%2);
         printf("Sono il FIGLIO 2.\n"); }
      exit(0);
   }
   // processo padre
   for (i=0; i<20; i++) {
      sleep(rand()%2);
      printf("Sono il PADRE.\n"); }
   exit(0);
}
```

Schedulazione dei processi

Abbiamo visto dall'esecuzione degli esempi precedenti che non si può determinare in maniera certa quale dei processi padre/figli (ma lo stesso discorso vale per tutti i processi del sistema) verranno eseguiti per prima. Inoltre in un sistema multi-tasking non ha senso parlare di prima e dopo: vengono eseguiti praticamente in parallelo.

Può essere desiderabile coordinare in qualche modo il lavoro del processo padre con quello del processo figlio: esiste una chiamata di sistema `pid_t wait(int *state)` che mette in attesa il processo corrente in attesa che termini almeno un processo figlio.
Viene restituito il PID del processo figlio appena terminato o `-1` in caso di errore. Il puntantore `int *state` punta ad un intero in cui verrà messo lo stato di uscita del processo che può eventualmente essere esaminato (se non ci interessa possiamo passare un valore `NULL`).
Se uno dei figli è già terminato, la chiamato ritorna subito. L'attesa può essere interrotta da un segnale (qualunque o di uscita).

Esempio di utilizzo di wait

forkwait.c

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

int main() {
   pid_t pid;
   int i;

   pid = fork();

   if (pid == 0) {
      // processo figlio
      for (i=0; i<10; i++) {
         sleep(rand()%2);
         printf("Sono il FIGLIO.\n");
      }
      exit(0);
   }

   sleep(rand()%2);
   // processo padre
   printf("Sono il processo PADRE e mi metto in attesa...\n");
   wait(NULL);
   printf("Sono il processo PADRE e ho terminato il mio lavoro.\n");
   exit(0);
}
```

Attesa di un processo specifico

Attraverso una chiamata del tipo `pid_t waitpid(pid_t pid, int *state, int options)` è possibile attendere l'esecuzione di un processo figlio specificato attraverso `pid`.
Il parametro `options` contiene alcune opzioni che possiamo anche trascurare (possiamo mettere `0`).
Anche qui possiamo mettere a `NULL` il parametro `state` se non ci interessa il suo stato di uscita.

Esempio di un *padre snaturato*...

multiforkwait.c

```
// creazione di piu' figli con l'attesa di uno specifico
#include <stdio.h>
#include <stdlib.h>
int main() {
   pid_t pid1,pid2;
   int i;
   pid1 = fork(); // creo un primo figlio
   if (pid1 == 0) {  // processo figlio (lento)
      for (i=0; i<15; i++) {
         sleep(rand()%4);
         printf("Sono il FIGLIO LENTO [%d].\n",getpid()); }
      printf("Sono il FIGLIO LENTO [%d] e ho finito.\n",getpid()); exit(0);
   }
   sleep(2);   // gli diamo un po' di vantaggio
   pid2 = fork(); // creo un secondo figlio
   if (pid2 == 0) {  // processo figlio (veloce)
      for (i=0; i<20; i++) {
         sleep(rand()%2);
         printf("Sono il FIGLIO VELOCE [%d].\n",getpid()); }
      printf("Sono il FIGLIO VELOCE [%d] e ho finito.\n",getpid()); exit(0);
   }
   // processo padre
   sleep(rand()%2);
   printf("Sono il processo PADRE e mi metto in attesa del figlio veloce [%d]...\n",pid2);
   waitpid(pid2,NULL,0);
   printf("Sono il processo PADRE e ho terminato il mio lavoro.\n");
   exit(0);
}
```

Esecuzione di altri programmi (1)

Nei sistemi UNIX è possibile da parte di un processo eseguire un programma esterno specificato. Questa operazione rimpiazza l'immagine del processo corrente (quello chiamante) con una immagine che eseguirà una istanza del programma specificato.
Abbiamo a disposizione una famiglia di chiamate del tipo `execX()`, dove la `X` può essere: `l`, `lp`, `v` o `vp`. Ognuna compie lo stesso compito ma in modo leggermente diverso.

Un comportamento comune è quello per cui, se il lancio del programma specificato non va a buon fine, allora viene riportato un errore e l'esecuzione del processo chiamante prosegue (magari stampando a video un errore). Se la messa in esecuzione va a buon fine, l'esecuzione passa all'istanza del programma specificato e non tornerà mai più al processo chiamante (che in effetti è stato sovrascritto).

Esecuzione di altri programmi (2)

In effetti il processo che esegue il programma specificato è esattamente quello che l'ha invocato: viene mantenuto pure il PID.
Un altro effetto della chiamata è quello di chiudere tutti i descrittori di file aperti dal processo chiamante ad eccezione di quelli predefiniti (input, output, error): ricordate l'esempio di redirezione dell'input con `dup()`?

Iniziamo con la chiamata:

```
int execl(char *path, [char *arg0, char *arg1,..., char *argN,] NULL)
```

Con `path` viene specificato il pathname del programma da eseguire, di seguito viene specificata una lista di puntatori a stringhe che rappresentano la linea di comando (il primo `arg0` è il nome con cui il programma *pensa* di essere stato invocato). Tale lista deve terminare con un parametro speciale di valore nullo.
La chiamata riporta `-1` in caso di errore e l'esecuzione prosegue.

Domanda: Cosa riporta in caso di buon fine?

Esecuzione di altri programmi (3)

La chiamata `execv()` ha una sintassi differente:

```
int execv(char *path, char *argv[])
```

In questo caso la linea di comando del programma da eseguire viene passata attraverso un array di stringhe in cui ogni stringa rappresenta un parametro (compreso sempre `arg0`).
L'array deve terminare con un ultimo puntatore a stringa nullo. Altrimenti come fa la chiamata a capire quando sono finiti i parametri?

Domanda

Qual'è la differenza sostanziale tra `execl()` ed `execv()`? Cosa permette di fare in più `execv()` che `execl()` non può fare?

Esecuzione di altri programmi (4)

Le varianti `execvp()` ed `execlp()` si comportano nello stesso modo delle controparti ad eccezione del fatto che se il primo parametro `pathname` in effetti non contiene dei caratteri / (quindi non si tratta di pathname!) cerca il programma da eseguire nella variabile d'ambiente `PATH`.
Quindi è molto utile quando dobbiamo eseguire dei comandi di sistema che si trovano nei percorsi predefiniti.

Esempio: invocare un comando

exec.c

```
// esegue un comando specificato come primo parametro
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
   if (argc == 1) {
      printf("utilizzo: %s comando\n",argv[0]);
      exit(1);
   }

   printf("Esegui il comando '%s'...\n",argv[1]);
   execlp(argv[1],argv[1],NULL);    // eseguo il comando passato come primo parametro

   // non e' necessario controllare se e' fallita...
   printf("L'invocazione di '%s' e' fallita!\n",argv[1]);   // ... lo e' di sicuro...
   // ... visto che il flusso di esecuzione e' arrivato fin qui.
   exit(1);
}
```

Esempio: una mini shell

nanoshell.c

```
// un abbozzo di funzionamento di una shell
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define LEN_BUFFER 1024
int main(int argc, char *argv[]) {
   char comando[LEN_BUFFER];
   int pid;
   while (1) {
      printf("Digitare un comando da eseguire ('quit' per uscire): ");
      scanf("%s",comando);
      if (strcmp(comando,"quit") == 0)
         break;
      pid = fork();
      if ( pid == -1 ) {
         printf("Errore nella fork.\n");
         exit(1);
      }
      if ( pid == 0 ) {
         execlp(comando, comando, NULL);
         printf("Errore nell'esecuzione di '%s'\n",comando);
         exit (2);
      } else
         wait(NULL);
   }
   exit(0);
}
```

Code di messaggi (1)

Nei sistemi UNIX i processi per scambiarsi i dati possono utilizzare dei costrutti di alto livello chiamati `code di messaggi`. In pratica, implementano delle code FIFO tra uno o più processi. Possono essere utilizzate dai processi per scambiarsi dati di ogni genere secondo uno schema produttore/consumatore. Volendo un processo può sia inserire che estrarre messaggi dalla coda, così come ci possono essere più di un processo ad inserire od estrarre da essa.

Le code di messaggi risiedono nella memoria centrale e non sul filesystem. Come concetto sono però simili ad un file: ci deve essere un processo che la crea ed un processo per leggervi/scrivervi deve prima aprirla. Vengono identificate da un numero intero positivo univoco che i processi scelgono per riferirsi alla stessa coda: viene detta chiave. Rappresenta il nome stesso della coda, così come il pathname serve ad identificare un file all'interno di un filesystem.

Code di messaggi (2)

Le code sono delle strutture permanenti, quindi *sopravvivono* al processo che le ha create. Bisogna cancellarle esplicitamente (così come i file).
Sono permanenti anche i messaggi inseriti e non ancora estratti.
I messaggi sotto UNIX non hanno delle strutture particolari, vengono trattati come sequenze non strutturate di byte.

Creazione/apertura di un coda (1)

Un processo può creare una nuova coda o aprirne una già esistente attraverso la chiamata:

```
int msgget(key_t key, int msgflg)
```

La chiamata crea o apre una coda di messaggi con chiave `key` e riporta un descrittore di coda che permette al processo di riferirsi a quella coda nelle successive operazioni (è un qualcosa di analogo ai descrittori di file). Se l'operazione non va a buon fine, viene riportato `-1`.

Il parametro `msgflg` specifica delle modalità di apertura (`IPC_CREAT` e `IPC_EXCL`) combinati attraverso l'operazione OR (`|`) con i permessi di accesso alla coda che eventualmente sarà creata. I permessi sono in formato ottale (tipo `0660`) con sintassi analoga a quanto visto per il filesystem (però l'esecuzione non ha al momento alcun senso). In caso di apertura i permessi specificati sono ignorati.

Creazione/apertura di un coda (2)

Se non esistono code con chiave `key` ed è stata specificata la modalità `IPC_CREAT`, allora la coda viene creata.
Se una coda con quella stessa chiave `key` dovesse già esistere in memoria, allora `msgget()` riporta un descrittore per quella stessa coda (la apre).

Se insieme al flag `IPC_CREAT` è stato attivato anche il flag `IPC_EXCL` allora il sistema si assicura che sia il processo chiamante ad aver creato la coda (e non un altro processo).

In caso di errore la chiamata ritorna `-1`.
L'uso delle code di messaggi richiede l'inclusione delle intestazioni: `sys/ipc.h` e `sys/msg.h`.

Come chiave `key` si può utilizzare il valore speciale `IPC_PRIVATE` per specificare una nuova coda non ancora creata.

Manipolazione di code

Per cancellare una coda, modificarne i permessi o raccogliere delle statistiche su di essa, si può usare la chiamata:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

Esegue il comando corrispondente a `cmd` sulla coda di descrittore `msqid`; le operazioni specificabili attraverso `cmd` sono:

- `IPC_RMID`: rimuove la coda associata a `msqid` (`buf` può essere `NULL`);
- `IPC_STAT`: raccoglie alcune statistiche sulla coda e le inserisce nella struttura puntata da `buf` (non scendiamo nei dettagli);
- `IPC_SET`: reimposta i diritti di accesso alla coda secondo quanto specificato nella struttura puntata da `buf` (non scendiamo nei dettagli).

Controllo delle risorse occupate

Può succedere che un processo allochi una risorsa persistente, come una coda di messaggi o una area di memoria condivisa, e che non la distrugga alla fine, magari per una sua terminazione imprevista.

Per controllare le risorse persistenti allocate si può usare il comando di shell `ipcs`: lista le risorse (code, segmenti condivisi e semafori) allocate in quel momento, con dati come la chiave, il proprietario e le modalità di accesso.

Per liberare una risorsa si può usare il comando `ipcrm [ shm|msg|sem ] id`, dove `id` è l'identificativo (non la chiave!) della risorsa.

Esempio: creazione e chiusura di una coda

msgcreate.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main() {
   int ds_coda;
   key_t chiave=40;

   printf("Creo la coda di messaggi di chiave %d... ",chiave);
   ds_coda= msgget(chiave, IPC_CREAT|IPC_EXCL|0660);
   if ( ds_coda == -1 ) {
      printf("\nErrore nella chiamata msgget().\n");
      exit(1);
   }
   printf("creata (descrittore %d)\n",ds_coda);
   sleep(5);
   printf("Chiudo la coda.\n");
   msgctl(ds_coda, IPC_RMID, NULL);
   exit(0);
}
```

Depositare messaggi nella coda

Una volta che un processo ha ottenuto il descrittore di coda (creandola o aprendola), vi può inserire dei messaggi arbitrari con:
`int msgsnd(int msqid, void *ptr, size_t size, int msgflg)`
Inserisce il messaggio puntato da `ptr` di dimensione `size` nella coda di messaggi individuata dal descrittore `msqid`. Il parametro `msgflg` specifica il comportamento della chiamata in caso di coda piena: `0` indica che la chiamata si deve bloccare se non c'è spazio nella coda in attesa che se ne liberi a sufficienza; con la modalità `IPC_NOWAIT` la chiamata ritorna subito un valore `-1` (così come in caso di errore) nel caso di coda piena.

E' importantissimo notare che in effetti il buffer puntato da `ptr` non ha una struttura del tutto libera: ogni messaggio ha un tipo indicato da un `long` positivo inserito all'inizio del buffer. Però la dimensione `size` si riferisce al contenuto del messaggio: quindi il buffer puntato da `ptr` deve puntare ad una struttura lunga `4+size` byte, in cui i primi 4 byte (un `long`) sono il tipo di messaggio.

Esempio: inserimento di un messaggio nella coda (1)

msgsend.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define DIM_MSG 1024

typedef struct{
   long mtype;
   char mtext[DIM_MSG];
} msg;

int main() {
   int ds_coda,risultato;
   key_t chiave=40;
   msg messaggio;

   // crea la coda
   ds_coda= msgget(chiave, IPC_CREAT|IPC_EXCL|0660);
   if ( ds_coda == -1 ) { printf("Errore nella chiamata msgget().\n"); exit(1); }

   // crea ed inserisce il messaggio
   messaggio.mtype=1;
   strncpy(messaggio.mtext,"Hello world!", DIM_MSG);
   risultato = msgsnd(ds_coda, &messaggio, strlen(messaggio.mtext)+1, IPC_NOWAIT);
```

Esempio: inserimento di un messaggio nella coda (2)

msgsend.c

```
   if (risultato == -1) {
      printf("Errore durante l'inserimento del messaggio.\n");
      exit(1);
   }
   msgctl(ds_coda, IPC_RMID, NULL);
   exit(0);
}
```

Prelevare messaggi dalla coda

Per leggere messaggi da una coda:

```
int msgrcv(int msqid, void *ptr, size_t size, long mtype,
int msgflg)
```

Legge dalla coda `msqid` un messaggio e lo scrive alla struttura puntata da `ptr`. Anche qui si tratta di una struttura che inizia con un `long` seguito da `size` byte destinati a contenere il contenuto del messaggio.

Se non ci sono messaggi da leggere la chiamata è bloccante per il processo chiamante a meno che non si utilizzi `IPC_NOWAIT` in `msgflg`.

Il parametro `mtype` serve ad estrarre messaggi di un certo tipo:

- 0: viene estratto il primo messaggio disponibile (a prescindere dal tipo) secondo l'ordine FIFO;
- >0: viene estratto il primo messaggio disponibile di tipo corrispondente a `mtype`;
- <0: viene estratto il primo tra i messaggi in coda di tipo `t`, con `t` minimo e con `t`$\leq$`|mtype|`.

Ritorna `-1` in caso di errore o il numero di byte del messaggio letto.

Esempio: estrazione di un messaggio dalla coda (1)

msgreceive.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define DIM_MSG 1024

typedef struct{
   long mtype;
   char mtext[DIM_MSG];
} msg;

int main() {
   int ds_coda;
   key_t chiave=40;
   msg messaggio;

   if ( (ds_coda= msgget(chiave, IPC_CREAT|IPC_EXCL|0660)) == -1) {
      printf("Errore nella chiamata msgget().\n");
      exit(1); }
   // crea ed invia il messaggio
   messaggio.mtype=1;
   strncpy(messaggio.mtext,"Hello world!", DIM_MSG);
   if (msgsnd(ds_coda, &messaggio, strlen(messaggio.mtext)+1, IPC_NOWAIT) == -1) {
      printf("Errore durante l'inserimento del messaggio.\n");
      exit(1); }
   strncpy(messaggio.mtext,"CANCELLO IL MESSAGGIO DAL BUFFER", DIM_MSG);
```

Esempio: estrazione di un messaggio dalla coda (2)

msgreceive.c

```
   // legge esso stesso il messaggio
   if (msgrcv(ds_coda, &messaggio, DIM_MSG, 0, 0) == -1) {
      printf("Errore durante l'estrazione del messaggio.\n");
      exit(1);
   } else
      printf("Messaggio letto: '%s'\n",messaggio.mtext);
   // chiudo la coda
   msgctl(ds_coda, IPC_RMID, NULL);
   exit(0);
}
```

Esempio: Produttore/Consumatore di stringhe (1)

msgtext.c

```
// Crea due processi figli: uno legge stringhe e le mette in coda,
// l'altro legge i messaggi dalla coda e li visualizza.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define DIM_MSG 1024
typedef struct{
   long mtype;
   char mtext[DIM_MSG];
} msg;

// legge le stringhe da tastiera e le mette in coda
void produttore(int coda) {
   msg messaggio;
   printf("Inserire le stringhe ('quit' per finire): \n");
   do {
      fgets(messaggio.mtext,DIM_MSG,stdin); // legge una riga (compresso l'invio)
      messaggio.mtype=1;
      if (msgsnd(coda, &messaggio, strlen(messaggio.mtext)+1, IPC_NOWAIT) == -1) {
         printf("Errore durante l'inserimento del messaggio.\n");
         exit(1); }
      printf("Messaggio inviato: %s\n",messaggio.mtext);
   } while ( strcmp(messaggio.mtext,"quit\n") != 0);
   exit(0);
}
```

Esempio: Produttore/Consumatore di stringhe (2)

msgtext.c

```
// riceve i messaggi dalla coda e li visualizza sullo standard output
void consumatore(int coda) {
   msg messaggio;
   do {
      // legge un messaggio bloccandosi se non ce ne sono
      if (msgrcv(coda, &messaggio, DIM_MSG, 1, 0) == -1) {
         printf("Errore durante l'estrazione del messaggio.\n");
         exit(1); }
      printf("Messaggio ricevuto: %s\n",messaggio.mtext);
   } while ( strcmp(messaggio.mtext,"quit\n") != 0);
   exit(0);
}
```

Esempio: Produttore/Consumatore di stringhe (3)

msgtext.c

```
int main() {
   int ds_coda;
   key_t chiave=IPC_PRIVATE;

   // crea la coda
   ds_coda= msgget(chiave, IPC_CREAT|IPC_EXCL|0660);
   if ( ds_coda == -1 ) { printf("Errore nella chiamata msgget().\n"); exit(1); }

   // crea i due processi figli: produttore e consumatore
   if ( fork() != 0) {
      if ( fork() != 0) {
         // corpo del padre
         wait(NULL);
         wait(NULL);
      } else
         produttore(ds_coda);
   } else
      consumatore(ds_coda);


   msgctl(ds_coda, IPC_RMID, NULL);
   exit(0);
}
```

Memoria condivisa (1)

Un altro costrutto che UNIX ci mette a disposizione per la comunicazione tra processi è la memoria condivisa. Una memoria condivisa è una porzione di memoria accessibile da più processi.
I processi che condividono la stessa area di memoria possono utilizzare tale zona per scambiarsi dei dati in modo arbitrario (leggendovi e scrivendovi).
Ogni memoria condivisa ha un nome univoco detto chiave di utilizzo del tutto analogo a quanto visto per le code di messaggi. La chiave è ciò che ogni processo utilizza per riferirsi alla medesima area di memoria da condividere.

Memoria condivisa (2)

Se un processo vuole usare un'area di memoria condivisa deve prima aprirla (così come per le code di messaggi) ed attaccarla al proprio spazio di indirizzamento. Una volta fatta questa operazione, può leggervi e scrivervi come una qualunque altra area disponibile, utilizzando le strutture dati più idonee.
Quando un processo non desidera più utilizzare quell'area di memoria condivisa può staccarla dal suo spazio di indirizzamento (ciò viene comunque fatto in fase di distruzione del processo). Questa operazione non implica la distruzione dell'area di memoria, infatti (così come per le code di messaggi) si tratta di strutture di memoria permanenti. E' necessario distruggere esplicitamente l'area di memoria (così come per le code di messaggi). Ovviamente, la proprietà di permanenza vale anche per i dati memorizzati nella memoria condivisa.

Creare/aprire/distruggere aree di memoria condivisa (1)

Con una sintassi/semantica simile a quanto visto per le code di messaggi, per creare/aprire una area di memoria condivisa si deve utilizzare la chiamata di sistema:

```
int shmget(key_t key, size_t size, int shmflg)
```

Dove `key` è la chiave per identificare l'area di memoria, `size` indica la taglia della memoria, ovvero quanto grande deve essere quest'area di memoria. I flag che si possono passare con `shmflg` sono analoghi a quanto visto per le code: una maschera ottale di diritti di accesso, combinati con flag del tipo `IPC_CREAT` e `IPC_EXCL`.

Come chiave `key` si può utilizzare il valore speciale `IPC_PRIVATE` per specificare una nuova area non già allocata.

Senza flag di apertura, la chiamata si aspetta che l'area già esista (ignorando quindi dimensione e diritti specificati); con `IPC_CREAT`, se non ne esiste una con quella chiave, la crea. Aggiungendo anche `IPC_EXCL` si impone una creazione esclusiva, quindi se già esiste viene riportato un errore.

Creare/aprire/distruggere aree di memoria condivisa (2)

La chiamata riporta un descrittore di memoria che servirà per la fase successiva. In caso di errore viene riportato `-1`.

E' interessante vedere che cosa succede alle aree di memoria condivise a seguito di alcune chiamate di sistema che hanno a che vedere con la gestione dei processi:

- `fork()`: il figlio eredita le aree di memoria condivise ed attaccate;
- `exec()` e `exit()` : tutte le aree di memoria condivise ed attaccate, vengono "staccate" (ma non distrutte).

Per distruggere un'area di memoria condivisa si deve usare esplicitamente la chiamata:

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

L'uso e i comandi di manipolazione sono analoghi a quanto visto per `msgctl()`, quindi ci limiteremo a dire che per l'eliminazione è sufficiente invocare: `shmctl(descrittore,IPC_RMID,NULL)`

Annettere un'area di memoria condivisa (1)

Per poter leggere e/o scrivere in una area di memoria che si è appena creata/aperta con `shmget()` è prima necessario chiedere al kernel di mappare tale area nel nostro spazio di indirizzamento, in modo tale che noi possiamo interagirvi come una qualunque altra area di memoria allocata. Per fare ciò, usiamo la chiamata:

```
void *shmat(int shmid, void *shmaddr, int shmflg)
```

Con `shmid` si indica il descrittore di memoria che ci ha riportato `msgget()`, `shmaddr` è un puntatore che indica l'indirizzo a cui vogliamo che l'area di memoria sia mappata oppure, passando `NULL`, lasciamo libero il kernel di sceglierci un qualunque indirizzo. In `shmflg` possiamo specificare il flag `SHM_RDONLY` per fare una mappatura in sola lettura (altrimenti viene mappato sia in lettura che in scrittura). Ci sono altri flag utili nel caso in cui specifichiamo un puntatore su cui fare la mappatura che non approfondiremo. Tutti questi flag si trovano nella intestazione `sys/shm.h`.

Annettere un'area di memoria condivisa (2)

La chiamata `shmat()` riporta l'indirizzo di memoria a cui è stata attaccata la memoria condivisa e a partire dal quale è possibile accedere alla memoria (come se fosse un array). Ricordatevi che le aree di memoria condivise sono sempre mappate in `modo contiguo`.
La chiamata riporta `-1` (un indirizzo che non esiste) in caso di errore.

Disnettere un'area di memoria collegata

Volendo staccare esplicitamente un'area di memoria già collegata al nostro spazio di indirizzamento si può usare la chiamata:

```
int shmdt(void *shmaddr)
```

Dove `shmaddr` è l'indirizzo a cui la memoria è stata in precedenza mappata.

Questa operazione non è, in effetti, strettamente necessaria: infatti quando il processo termina, il suo spazio di indirizzamento viene distrutto e quindi la memoria condivisa viene "staccata". Questo però non implica che l'area di memoria condivisa venga distrutta (continua ad esistere).

Esempio: Scambio di numeri con memoria condivisa (1)

shmnumbers.c

```
// Crea un processo figlio che legge numeri da tastiera e li deposita in
// un'area di memoria condivisa; poi ne crea un altro che li rilegge.
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MAX_NUMBERS  10

// legge numeri da tastiera mettendoli nella memoria condivisa
void scrittore(int id) {
   int *p;
   int numero,q=0;
   printf("\nProcesso Scrittore:\n");
   // collega la memoria condivisa con lo spazio di indirizzamento (in lettura/scrittura)
   p= (int *) shmat(id,NULL,0);
   if ( p == (int *)-1 ) { printf("Errore nella chiamata shmat().\n"); exit(1); }
   do {
      printf("\nNumero da inserire ('0' per finire): ");
      scanf("%d",&numero);
      if (numero==0) break;
      p[q+1]=numero; // scrive nella memoria (lasciando il primo slot libero)
      q++;
      printf("Numero inserito: %d\n",numero);
   } while ( q < MAX_NUMBERS );
   p[0]=q;     // scrive nel primo slot, la quantita' di numeri depositati
   exit(0); // il primo figlio esce
}
```

Esempio: Scambio di numeri con memoria condivisa (2)

shmnumbers.c

```
// prende i numeri dalla memoria condivisa e li stampa a video
void lettore(int id) {
   int *p;
   int i,q=0;
   printf("\nProcesso Lettore:\n");
   // collega la memoria condivisa con lo spazio di indirizzamento (in sola lettura)
   p= (int *) shmat(id,NULL,SHM_RDONLY);
   if ( p == (int *)-1 ) { printf("Errore nella chiamata shmat().\n"); exit(1); }
   q=p[0];     // legge quanti numeri ci sono nell'area condivisa
   printf("n.%d numeri depositati: ",q);
   for (i=1;i<=q;i++)
      printf("%d ",p[i]);
   printf("\n");
   exit(0); // il secondo figlio esce
}
```

Esempio: Scambio di numeri con memoria condivisa (3)

shmnumbers.c

```
int main() {
   int id_shm;

   // crea l'area di memoria condivisa
   id_shm=shmget(IPC_PRIVATE,sizeof(int)*(MAX_NUMBERS+1),IPC_CREAT|IPC_EXCL|0600);
   if ( id_shm == -1 ) { printf("Errore nella chiamata shmget().\n"); exit(1); }

   // crea un primo processo figlio che scrive nella memoria condivisa
   if ( fork() != 0)
      wait(NULL);
   else
      scrittore(id_shm);

   // dopo che il primo figlio e' uscito, il secondo legge dalla memoria condivisa
   if ( fork() != 0)
      wait(NULL);
   else
      lettore(id_shm);

   // distrugge l'area di memoria condivisa
   shmctl(id_shm, IPC_RMID, NULL);
   exit(0);
}
```

Modalità di utilizzo dell'area condivisa

Tra i meccanismi di comunicazione tra processi, l'uso di un'area di memoria condivisa è il metodo più efficiente.
Inoltre le code di messaggi vincolano la trasmissione in modo sequenziale, limitandone l'utilità in certi contesti di uso.
Nell'esempio precedente, il processo scrittore inserisce i dati nell'area condivisa e, solo dopo che questo ha finito, il processo lettore estrae i dati.
Questa modalità di utilizzo non crea problemi.
I problemi sorgono quando i due processi cercano di accedervi (in lettura/scrittura) in modo concorrente.

Esempio di problema nella modifica concorrente

Abbiamo già visto nella teoria i problemi legati alle corse critiche.
Riprendiamo qui un esempio: supponiamo che `P1` e `P2` siano due processi di gestione per il versamento su un conto corrente, dove `x` è il saldo del conto stesso.
Una operazione di incremento del tipo `x=x+1` non è atomica, infatti può essere vista come una sequenza:

- leggi `x` dalla memoria condivisa;
- calcola `x+1`;
- scrivi il nuovo valore nella memoria condivisa.

Potrebbe succedere che `P1`, incrementando `x`, venga interrotto dallo scheduler a metà della sequenza (poco prima di scrivere il nuovo valore).
A questo punto il controllo passa a `P2` che completa il suo incremento.
Una volta ripreso il controllo `P1`, questo finisce la sua operazione con il risultato che il versamento fatto da `P2` viene "sovrascritto".
Questo crea una inconsistenza nelle operazioni.

Esempio: problema con la concorrenza (1)

concproblem.c

```
// Un esempio che mostra i problemi con le esecuzioni concorrenti:
// entrambi i figli incrementano la variabile condivisa, controllandone i risultati finali
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main() {
   int id_shm;
   int *conto;
   int temp,versamento=1000;

   id_shm=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|IPC_EXCL|0600);
   if ( id_shm == -1 ) { printf("Errore nella chiamata shmget().\n"); exit(1); }

   conto= (int *) shmat(id_shm,NULL,0);   // i figli erediteranno l'area condivisa
   *conto=0;

   if ( fork() == 0) {     // primo figlio
      for (;versamento>0;versamento--){
         //(*conto)++;
         temp=*conto;
         printf("P1: letto=%d\t scritto=%d\n",temp,temp+1);
         *conto=temp+1;
         usleep(rand()%5000);
      }
```

Esempio: problema con la concorrenza (2)

concproblem.c

```
      exit(0);
   } else if ( fork() == 0) {    // secondo figlio
      for (;versamento>0;versamento--){
         //(*conto)++;
         temp=*conto;
         printf("P2: letto=%d\t scritto=%d\n",temp,temp+1);
         *conto=temp+1;
         usleep(rand()%5000);
      }
      exit(0);
   }

   wait(NULL); wait(NULL); // aspetta che finiscano entrambi i figli
   printf("Padre: valore finale=%d\n",*conto);

   shmctl(id_shm, IPC_RMID, NULL);
   exit(0);
}
```

Sezioni critiche

Il problema visto negli esempi precedenti si può generalizzare e risolvere definendo le cosiddette sezioni critiche: consideriamo un sistema in cui ci sono `n` processi che competono per l'utilizzo di alcune risorse condivise (per esempio, aree di memoria, tabelle, file, dispositivi, ecc.).
Per ogni processo che ha la necessità di accedere in scrittura alla risorsa condivisa, esiste una porzione di codice detta sezione critica che raccoglie le istruzioni di alterazione della risorsa.
Affinché non si presentino problemi di inconsistenza dello stato della risorsa si deve fare in modo che quando un processo entra nella sua sezione critica, nessun altro processo sia autorizzato ad entrare nella propria. In altre parole, le sezioni critiche dei vari processi devono essere eseguite in mutua esclusione.
Questo è sufficiente per garantire la coerenza delle operazioni di accesso.

I semafori

Un modo semplice per risolvere il problema delle sezioni critiche è quello di impiegare i semafori.

Un semaforo non è altro che una variabile numerica intera `S`, contenente un valore non negativo, a cui si può accedere solo con due operazioni atomiche standard denominate `WAIT` e `SIGNAL`:

- `WAIT`: se `S`>0, allora decrementa `S` di una unità; se `S` ha valore nullo, allora mette il processo in attesa che il valore diventi strettamente positivo;
- `SIGNAL`: incrementa `S` di una unità.

Le operazioni sono atomiche nel senso che il sistema operativo deve garantire che l'esecuzione delle istruzioni che compongono tali operazioni non può essere interrotta a metà dallo scheduler per l'esecuzione di un altro processo (concorrente).

I semafori per la mutua esclusione

Per risolvere il problema delle sezioni critiche si può utilizzare un semaforo `S` inizializzato a `1` e sfruttare le operazioni nel seguente modo:

- `WAIT`: da chiamare quando il processo entra nella sua sezione critica;
- `SIGNAL`: da usare quando si esce dalla sezione critica.

In questo modo, quando non c'è nessun processo nella sua sezione critica il contatore `S` ha sempre valore `1`. Quando un processo è nella sua sezione critica, l'operazione `WAIT` azzera `S` e vieta a qualunque altro processo di entrare nella propria sezione critica. Solo quando il processo esce dalla sua sezione critica `S` tornerà positivo e, se c'era un processo concorrente bloccato in attesa, questo verrà sbloccato.

I semafori per il conteggio delle risorse (1)

I semafori si possono utilizzare anche per risolvere il problema del produttore/consumatore: supponiamo di avere un processo produttore `P` che produce dei dati che inserisce in una coda di dimensione `n` e di avere un processo consumatore `C` che preleva questi dati e che si deve bloccare in attesa se la coda è vuota.

Per implementare questo meccanismo di blocco si può utilizzare un semaforo `S1` inizializzato a zero: il produttore `P` dopo aver depositato un dato nella coda utilizza `SIGNAL` per incrementare il contatore `S1`, il consumatore prima di tentare di prelevare un dato dalla coda, invoca `WAIT`; ciò implica un bloccaggio di `C` se la coda è vuota.

Praticamente `S1` conteggia le posizioni occupate nella coda.

I semafori per il conteggio delle risorse (2)

In realtà esiste anche il problema duale per il produttore nel caso in cui la coda sia di dimensione finita: `P` si deve bloccare nell'inserimento se non c'è spazio nella coda, in attesa che `C` estragga dei dati.
Ciò viene implementato con un secondo semaforo `S2` inizializzato a `n` (la dimensione della coda): `P` invoca `WAIT` su `S2` prima di un inserimento ed il consumatore `C` utilizza `SIGNAL` subito dopo aver fatto l'estrazione.
Praticamente `S2` conteggia le posizioni libere nella coda.

Per garantire una perfetta implementazione di un meccanismo produttore/consumatore si deve utilizzare un ulteriore semaforo `S3` per la mutua esclusione delle operazioni sulla coda (per un totale di 3 semafori).

Questo meccanismo in effetti si può utilizzare anche con più produttori e più consumatori.

I Semafori sotto Unix

I sistemi UNIX offrono una implementazione dei semafori che è possibile utilizzare nei nostri programmi. In realtà, vengono gestiti array di semafori. Ogni semaforo però può essere gestito individualmente.
Per creare/aprire un array di semafori si usa la chiamata:
`int semget(key_t key, int nsems, int semflg)`
Il significato è molto simile a quanto visto per le code e per le aree di memoria condivise: `key` è una chiave numerica che identifica l'array e `semflg` permette di specificare le modalità di creazione/apertura con i soliti flag `IPC_CREAT` e `IPC_EXCL`. Per `key` si può usare anche la costante `IPC_PRIVATE`. `nsems` specifica il numero di semafori che l'array deve gestire.
In caso di successo, viene riportato un descrittore di semafori (o identificatore di semafori) che individua l'array di semafori all'interno del sistema. In caso di problemi viene riportato `-1`.

Operazioni sui semafori (1)

Per operare sul contatore di un semaforo (di un array) si deve usare la chiamata:

```
int semop(int semid, struct sembuf *ops, unsigned nops)
```

`semid` è il descrittore riportato da `semget()` che identifica l'array di semafori su cui operare, `ops` punta ad un vettore formato da una o più istanze di una struttura standard definita in `sys/sem.h`. Ogni elemento di questo vettore rappresenta una operazione da compiere su un semaforo dell'array. `nops` contiene il numero di elementi del vettore puntato da `ops`.

Ogni operazione di `ops` opera su un singolo semaforo. L'insieme delle operazioni specificate con `ops` viene eseguito in `modo atomico`.
Per semplificare la spiegazione, supponiamo di voler fare una operazione alla volta, quindi supporremo che `ops` punti ad una singola struttura e che `nops` sia pari a `1`.

Operazioni sui semafori (2)

La struttura `struct sembuf` è formata dai seguenti campi:

- `sem_num`: il numero del semaforo dell'array su cui operare (la numerazione parte da `0`);
- `sem_op`: indica l'operazione da effettuare;
- `sem_flg`: specifica la politica di attesa nel caso di più operazioni (ad esempio si può utilizzare `IPC_NOWAIT`), ma per semplicità assumeremo nel seguito che `sem_flg=0`.

Supponiamo che il valore del semaforo numero `sem_num` sia `S`:

- `sem_op<0`: se `S+sem_op<0`, allora il processo chiamante viene sospeso finché `S+sem_op`$\geq$`0`, nel qual caso viene aggiunto il valore negativo `sem_op` ad `S` (che quindi continuerà ad essere non negativo);
- `sem_op=0`: il processo chiamante viene sospeso finché `S=0`;
- `sem_op>0`: viene sommato `sem_op` ad `S`.

Vedremo come utilizzare `semop()` per implementare `WAIT` e `SIGNAL`.

Implementare WAIT e SIGNAL

Il kernel non mette a disposizione le operazioni studiate nella teoria di WAIT e SIGNAL, ma si possono ottenere in modo molto semplice:

```
int WAIT(int sem_des, int num_semaforo){
   struct sembuf operazioni[1] = {{num_semaforo,-1,0}};
   return semop(sem_des, operazioni, 1);
}

int SIGNAL(int sem_des, int num_semaforo){
   struct sembuf operazioni[1] = {{num_semaforo,+1,0}};
   return semop(sem_des, operazioni, 1);
}
```

Inizializzare e leggere un semaforo (1)

Prima di porter iniziare ad utilizzare i semafori è necessario imparare ad impostare il loro valore iniziale. Per fare ciò esiste una chiamata apposita abbastanza complessa che permette di fare anche molte altre cose. Noi ne vedremo solo alcuni dettagli che ci mettano in grado di utilizzare i semafori sotto UNIX in modo proficuo:

```
int semctl(int sem_des, int sem_num, int cmd, union senum arg)
```

Il valore `sem_des` individua l'array di semafori e `sem_num` seleziona uno specifico semaforo su cui operare (la numerazione parte sempre da `0`).
Il campo `arg` rappresenta gli argomenti dell'operazione e variano da un'operazione all'altra. Si tratta di una `union` che raccoglie i vari tipi richiesti dalle operazioni. Semplificandola solo alle operazioni che vedremo, abbiamo:

```
union semun {
    int val;  /* usato se cmd == SETVAL */
    unsigned short *array;  /* se cmd == SETALL */
}
```

Inizializzare e leggere un semaforo (2)

Tra le operazioni che possiamo specificare attraverso `cmd` abbiamo:

- `IPC_RMID`: rimuove l'array di semafori indicato da `sem_des`; tutti gli altri argomenti vengono ignorati;
- `SETVAL`: il valore in `val` (elemento della union `semun`) viene assegnato al semaforo numero `sem_num`;
- `SETALL`: tutti i semafori dell'array `sem_des` vengono inizializzati con i valori contenuti nel vettore puntato dall'elemento della union `array` (che il programmatore deve predisporre); il primo semaforo prenderà il valore `array[0]`, il secondo `array[1]`, ecc.

Ci sono altre operazioni che tralasciamo per semplicità.

Esempio: versamenti concorrenti con semaforo (1)

concfixed.c

```
// L'esempio del problema con i versamenti concorrenti risolto con un semaforo.
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

int main() {
   int id_shm, id_sem;
   int *conto;
   int temp,versamento=1000;
   struct sembuf WAIT[1]={{0,-1,0}};
   struct sembuf SIGNAL[1]={{0,+1,0}};

   id_shm=shmget(IPC_PRIVATE,sizeof(int),IPC_CREAT|IPC_EXCL|0600);
   if ( id_shm == -1 ) { printf("Errore nella chiamata shmget().\n"); exit(1); }
   conto= (int *) shmat(id_shm,NULL,0);   // i figli erediteranno l'area condivisa
   *conto=0;

   id_sem= semget(IPC_PRIVATE, 1, IPC_CREAT|IPC_EXCL|0600);
   if ( id_sem == -1 ) { printf("Errore nella chiamata semget().\n"); exit(1); }
   // imposta il valore iniziale del semaforo ad 1 (semaforo mutex)
   if ( semctl(id_sem, 0, SETVAL, 1) == -1 ) { printf("Errore nella chiamata semctl().\n"); e
```

Esempio: versamenti concorrenti con semaforo (2)

concfixed.c

```
   if ( fork() == 0) {      // primo figlio
      for (;versamento>0;versamento--){
         semop(id_sem,WAIT,1);   // inizio sezione critica: WAIT
         temp=*conto;
         printf("P1: letto=%d\t scritto=%d\n",temp,temp+1);
         *conto=temp+1;
         semop(id_sem,SIGNAL,1); // fine sezione critica: SIGNAL
         usleep(rand()%5000);
      }
      exit(0);
   } else if ( fork() == 0) {    // secondo figlio
      for (;versamento>0;versamento--){
         semop(id_sem,WAIT,1);   // inizio sezione critica: WAIT
         temp=*conto;
         printf("P2: letto=%d\t scritto=%d\n",temp,temp+1);
         *conto=temp+1;
         semop(id_sem,SIGNAL,1); // fine sezione critica: SIGNAL
         usleep(rand()%5000);
      }
      exit(0);
   }

   wait(NULL); wait(NULL); // aspetta che finiscano entrambi i figli
   printf("Padre: valore finale=%d\n",*conto);
   shmctl(id_shm, IPC_RMID, NULL);
   semctl(id_sem, 0, IPC_RMID, 0);
   exit(0);
}
```

Esempio: prod./cons. con memoria cond. e semafori (1)

shmsemprodcons.c

```
// Due figli: un produttore ed un consumatori di numeri generati a random.
// I due processi usano una area di memoria condivisa che contiene piu'
// elementi e 3 semafori per coordinarsi.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define DIM_BUFFER   10
#define HOW_MANY  200
#define S_MUTEX   0
#define S_EMPTY   1
#define S_FULL  2

int WAIT(int sem_des, int num_semaforo){
   struct sembuf operazioni[1] = {{num_semaforo,-1,0}};
   return semop(sem_des, operazioni, 1);
}

int SIGNAL(int sem_des, int num_semaforo){
   struct sembuf operazioni[1] = {{num_semaforo,+1,0}};
   return semop(sem_des, operazioni, 1);
}
```

Esempio: prod./cons. con memoria cond. e semafori (2)

shmsemprodcons.c

```
// produce numeri random e li scrive nella memoria condivisa
void produttore(int shm, int sems) {
   int *p,*in,*out;
   int i,number,len;
   long tot=0;
   srand(time(NULL));   // randomizza il generatore di numeri pseudo-causali
   p= (int *) shmat(shm, NULL, 0);  // attacca la memoria condivisia
   if ( p == (int *)-1 ) { printf("Errore nella chiamata shmat().\n"); exit(1); }
   in=p+DIM_BUFFER;  // la penultima posizione conterra' l'indice dell'inizio coda
   out=p+DIM_BUFFER+1;  // l'ultima posizione conterra' l'indice della fine della coda
   *in=0; *out=0;
   for (i=0;i<HOW_MANY;i++) {
      WAIT(sems,S_EMPTY);  // diminuisce le posizioni vuote
      WAIT(sems,S_MUTEX);  // entra nella sezione critica

      number=rand()%500;
      p[*in]=number;       // scrive il numero nella coda
      *in=(*in+1)%DIM_BUFFER; // sposta l'indice in avanti (ciclicamente)
      len=(*in>*out? (*in-*out) : (DIM_BUFFER - (*out-*in)) ); // calcola in # di elementi in
      printf("Produttore: %d  \t(in=%d out=%d len=%d)\n",number,*in,*out,len);
      tot+=number;

      SIGNAL(sems,S_MUTEX);   // esce dalla sezione critica
      SIGNAL(sems,S_FULL); // incrementa le posizioni piene
      usleep(rand()%40000);
   }
   printf("Produttore: totale finale=%ld\n",tot);
   exit(0);
}
```

Esempio: prod./cons. con memoria cond. e semafori (3)

shmsemprodcons.c

```
// legge i numeri dalla coda e li stampa a video
void consumatore(int shm, int sems) {
   int *p,*in,*out;
   int i,number,len;
   long tot=0;

   p= (int *) shmat(shm, NULL, 0); // attacca la memoria condivisia
   if ( p == (int *)-1 ) { printf("Errore nella chiamata shmat().\n"); exit(1); }
   in=p+DIM_BUFFER;  // la penultima posizione conterra' l'indice dell'inizio coda
   out=p+DIM_BUFFER+1;  // l'ultima posizione conterra' l'indice della fine della coda

   for (i=0;i<HOW_MANY;i++) {
      WAIT(sems,S_FULL);   // diminuisce le posizioni piene
      WAIT(sems,S_MUTEX);  // entra nella sezione critica

      number=p[*out];          // legge il numero dalla coda
      *out=(*out+1)%DIM_BUFFER;  // sposta l'indice in avanti (ciclicamente)
      len=(*in>=*out? (*in-*out) : (DIM_BUFFER - (*out-*in)) );   // calcola in # di elementi
      printf("Consumatore: %d  \t(in=%d out=%d len=%d)\n",number,*in,*out,len);
      tot+=number;

      SIGNAL(sems,S_MUTEX);   // esce dalla sezione critica
      SIGNAL(sems,S_EMPTY);   // incrementa le posizioni vuote
      usleep(rand()%50000);
   }
   printf("Consumatore: totale finale=%ld\n",tot);
   exit(0);
}
```

Esempio: prod./cons. con memoria cond. e semafori (4)

shmsemprodcons.c

```
int main() {
   int id_shm, id_sems;

   // crea l'area di memoria condivisa per i DIM_BUFFER numeri e due variabili aggiuntive
   id_shm= shmget(IPC_PRIVATE, (DIM_BUFFER+2)*sizeof(int), IPC_CREAT|IPC_EXCL|0600);
   if ( id_shm == -1 ) { printf("Errore nella chiamata shmget().\n"); exit(1); }
   // crea i 3 semafori (S_MUTEX,S_EMPTY,S_FULL)
   id_sems= semget(IPC_PRIVATE, 3, IPC_CREAT|IPC_EXCL|0600);
   if ( id_sems == -1 ) { printf("Errore nella chiamata semget().\n"); exit(1); }
   // imposta i valori iniziali dei semafori
   semctl(id_sems, S_MUTEX, SETVAL, 1);
   semctl(id_sems, S_EMPTY, SETVAL, DIM_BUFFER);
   semctl(id_sems, S_FULL, SETVAL, 0);
   // crea i due processi figli: produttore e consumatore
   if ( fork() != 0) {
      if ( fork() != 0) {
         // corpo del padre
         wait(NULL);
         wait(NULL);
      } else
         produttore(id_shm,id_sems);
   } else
      consumatore(id_shm,id_sems);
   // distrugge memoria condivisa e semaforo
   shmctl(id_shm, IPC_RMID, NULL);
   semctl(id_sems, 0, IPC_RMID, 0);
   exit(0);
}
```

I Segnali sotto UNIX

I segnali sono un semplice mezzo per notificare ai processi degli eventi asincroni. Si possono vedere come degli interrupt software. A differenza dei messaggi delle code FIFO:

- un segnale può essere inviato in qualsiasi momento, occasionalmente da un processo, ma spesso dal kernel (ad esempio, per una istruzione illegale);
- un segnale non viene necessariamente ricevuto e processato; implicitamente la maggior parte dei segnali provoca la terminazione del processo. Eventualmente un processo può decidere di ignorarli o gestirli;
- i segnali non hanno alcun contenuto informativo; in particolare, non si può conoscere il mittente del segnale (potrebbe essere il kernel o un altro processo).

Segnali più comuni (1)

Di seguito riportiamo una lista dei segnali principali con relative costanti:

- `SIGHUP` (1): hangup. Un processo riceve questo segnale quando il terminale a cui era associato viene chiuso o scollegato (la finestra viene chiusa o, nel caso di un collegamento remoto, la connessione cade).
- `SIGINT` (2): Interrupt. Viene ricevuto da un processo quando l'utente preme la combinazione di tasti di interrupt (solitamente `CTRL+C`);
- `SIGQUIT` (3): Quit. Simile a `SIGINT` ad eccezione del fatto che il processo genera un code dump, ovvero un file in cui viene messa l'immagine della memoria del processo al momento in cui si riceve il segnale `SIGQUIT`. Questa immagine può essere utile ai fini del debugging. Solitamente questo segnale viene invocato dalla combinazione di tasti `CTRL+\`;
- `SIGILL` (4): Illegal Instruction. Il processo ha tentato di eseguire una istruzione proibita o inesistente;
- `SIGKILL` (9): questo segnale non può essere intercettato dal processo, che non può fare altro che terminare. E' il modo più certo e brutale per "uccidere" un processo;

Segnali più comuni (2)

- `SIGSEGV` (11): Segmentation Violation. Generato quando il processo tenta di accedere ad un indirizzo che ricade fuori dalle aree di memoria allocate;
- `SIGALRM` (14): Alarm. Un segnale che il processo si può auto-inviare alla scadenza di un certo intervallo di tempo;
- `SIGCHLD` (17): Child Death. Inviato ad un processo quando uno dei suoi figli termina.

Per ulteriori segnali e dettagli consultare: `man 7 signal`.
Tutte queste costanti sono definite nell'header `signal.h`.

Gestione dei segnali

I processi possono decidere di ignorare o di "armare" (collegare una propria procedura che verrà invocata ogni volta che il nostro processo riceve tale segnale) alcuni segnali. Fanno eccezione i segnali di errore fatale (4, 11) o il segnale di kill (9) che non possono in alcun modo essere gestiti e che portano alla terminazione del processo.

Inviare un segnale

Per inviare un segnale ad un altro processo si può utilizzare la chiamata di sistema:

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

Dove `pid` è il PID del processo a cui vogliamo inviare il segnale di numero `sig`.

La chiamata `kill()` ritorna `-1` in caso di errore, `0` in caso di successo.

Un processo può inviare segnali solo ad altri processi che appartengono allo stesso proprietario. Ad esempio, non possiamo uccidere i processi di un altro utente. A tutto questo fa eccezione l'amministratore root, lui può tutto...