Programmazione di Sistema in UNIX

Nicola Drago

Università di Verona
Dip. di Informatica
Verona

Sommario

- Interfaccia tramite system call
- System call:
  - Gestione di file
  - Gestione di processi
  - Comunicazione tra processi

Interfaccia tramite system call

- L'accesso al kernel è permesso soltanto tramite le system call, che permettono di passare all'esecuzione in modo kernel.
- Dal punto di vista dell'utente, l'interfaccia tramite system call funziona come una normale chiamata C.
- In realtà più complicato:
  - Esiste una *system call library* contenente funzioni con lo stesso nome della system call
  - Le funzioni di libreria cambiano il modo user in modo kernel e fanno sì che il kernel esegua il vero e proprio codice delle system call
  - La funzione di libreria passa un identificatore, unico, al kernel, che identifica una precisa system call.
  - Simile a una routine di interrupt (detta *operating system trap*)

System Call

Classe	System Call	
File	`creat()`	`open()`
	`close()`	`read()`
	`write()`	`creat()`
	`lseek()`	`dup()`
	`link()`	`unlink()`
	`stat()`	`fstat()`
	`chmod()`	`chown()`
	`umask()`	`ioctl()`
Processi	`fork()`	`exec()`
	`wait()`	`exit()`
	`signal()`	`kill()`
	`getpid()`	`getppid()`
	`alarm()`	`chdir()`
Comunicazione tra processi	`pipe()`	`msgget()`
	`msgctl()`	`msgrcv()`
	`msgsnd()`	`semop()`
	`semget()`	`shmget()`
	`shmat()`	`shmdt()`

Efficienza delle system call

- L'utilizzo di system call è in genere meno efficiente delle (eventuali) corrispondenti chiamate di libreria C
- Particolarmente evidente nel caso di system call per il file system
  - Esempio:

```
/* PROG1 */
int main(void) {
    int c;
    while ((c = getchar()) != EOF) putchar(c);
}
/* PROG2 */
int main(void) {
    char c;
    while (read(0, &c, 1) > 0)
        if (write(1, &c, 1) != 1) perror("write"), exit(1);
}
```

    PROG1 è circa 5 volte più veloce!

Errori nelle chiamate di sistema

- In caso di errore, le system call ritornano tipicamente un valore -1, ed assegnano lo specifico codice di errore nella variabile `errno`, definita in `<errno.h>`
- Per mappare il codice di errore al tipo di errore, si utilizza la funzione

```
#include <stdio.h>
void perror (char *str)
```

  su `stderr` viene stampato:

$$str \texttt{ : } messaggio\text{-}di\text{-}errore \texttt{ \textbackslash n}$$

- Solitamente *str* è il nome del programma o della funzione.
- Per comodità definiamo una funzione di errore alternativa `syserr`, definita in un file `mylib.c`
  - Tutti i programmi descritti di seguito devono includere `mylib.h` e llinkare `mylib.o`

```
/*****************************************
MODULO: mylib.h
SCOPO: definizioni per la libreria mylib
*****************************************/

void syserr (char *prog, char *msg);
```

```
/*****************************************
MODULO: mylib.c
SCOPO: libreria di funzioni d'utilita'
*****************************************/
#include <stdio.h>
#include <errno.h>

#include "mylib.h"

void syserr (char *prog, char *msg)
{
  fprintf (stderr, "%s - errore: %s\n",prog, msg);
  perror ("system error");
  exit (1);
}
```

System Call per il File System

Introduzione

- In UNIX esistono quattro tipi di file
  1. File regolari
  2. Directory
  3. *pipe* o *fifo*
  4. *special file*
- Gli special file rappresentano un device (*block device* o *character device*)
- Non contengono dati, ma solo un puntatore al device driver:
  - *Major number*: indica il tipo del device (driver)
  - *Minor number*: indica il numero di unità del device

I/O non bufferizzato

- Le funzioni in `stdio.h` sono tutte bufferizzate. Per efficienza, si può lavorare direttamente sui buffer.
- In questo caso i file non sono più descritti da uno *stream* ma da un *descrittore* (un intero piccolo).
- Alla partenza di un processo, i primi tre descrittori vengono aperti automaticamente dalla shell:

```
0 ... stdin
1 ... stdout
2 ... stderr
```

- Per distinguere, si parla di *canali* o *stream* anziché di file.

Apertura di un canale

```
#include <fcntl.h>

int open (char *name, int access, mode_t mode)
```

Valori del parametro `access`:

- uno a scelta fra:
  `O_RDONLY O_WRONLY O_RDWR`
- uno o più fra:
  `O_APPEND O_CREAT O_EXCL O_SYNC O_TRUNC`

Valori del parametro `mode`: uno o più fra i seguenti:

`IRUSR IWUSR IXUSR IRGRP IWGRP IXGRP IROTH IWOTH IXOTH`

Corrispondenti ai modi di un file UNIX (u=RWX,g=RWX,o=RWX), e rimpiazzabili dai codici numerici (`000...777`)

Apertura di un canale (2)

- Modi speciali di `open`:
  - `O_EXCL`: apertura in modo esclusivo (nessun altro processo può aprire)
  - `O_SYNC`: apertura in modo sincronizzato (file tipo lock)
  - `O_TRUNC`: apertura di file esistente implica cancellazione
- Esempi di utilizzo:
  - `int fd = open("file.dat",O_RDONLY|O_EXCL,IRUSR|IRGRP|IROTH);`
  - `int fd = open("file.dat",O_CREAT, IRUSR|IWUSR|IXUSR);`
  - `int fd = open("file.dat",O_CREAT, 700);`

Apertura di un canale (3)

```
#include <fcntl.h>

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

- `creat` crea un file (più precisamente un inode) e lo apre in lettura.
  - Parametro `mode`: come `access`
- Sebbene `open` sia usabile per creare un file, tipicamente si utilizza `creat` per creare un file, e la `open` per aprire un file esistente da leggere/scrivere.

Creazione di una directory

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

int mknod(char *path, mode_t mode, dev_t dev)
```

- Simile a `creat`: crea un i-node per un file
- Può essere usata per creare un file
- Più tipicamente usata per creare directory e special file
- Solo il super-user può usarla (eccetto che per special file)

Creazione di una directory – (cont.)

- Valori di `mode`:
  - Per indicare tipo di file:

`S_IFIFO`	0010000	FIFO special
`S_IFCHR`	0020000	Character special
`S_IFDIR`	0040000	Directory
`S_IFBLK`	0060000	Block special
`S_IFREG`	0100000	
	0000000	Ordinary file

  - Per indicare il modo di esecuzione:

`S_ISUID`	0004000	et user ID on execution
`S_ISGID`	0002000	Set group ID on execution
`S_ISVTX`	0001000	Save text image after execution

Creazione di una directory – (cont.)

- Per indicare i permessi:

S_IREAD	0000400	Read by owner
S_IWRITE	0000200	Write by owner
S_IEXEC	0000100	Execute (search on directory) by owner
s_IRWXG	0000070	Read, write, execute (search) by group
S_IRWXD	0000007	Read, write, execute (search) by others

- il parametro `dev` indica il major e minor number del device, mentre viene ignorato se non si tratta di uno special file.

Creazione di una directory – (cont.)

- La creazione con `creat` di una directory NON genera le entry "." e ".."
- Queste devono essere create "a mano" per rendere usabile la directory stessa.
- In alternativa (consigliato) si possono utilizzare le funzioni di libreria:

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

int mkdir (const char *path, mode_t mode);
int rmdir (const char *path);
```

Manipolazione diretta di un file

```
#include <unistd.h>
ssize_t read  (int fildes, void *buf, size_t n)
ssize_t write (int fildes, void *buf, size_t n)
int close (int fildes)

#include <sys/types.h>
#include <unistd.h>
off_t  lseek (int fildes, off_t o, int whence)
```

- Tutte le funzioni restituiscono -1 in caso di errore.
- n: numero di byte letti. Massima efficienza quando $n =$ dimensione del blocco fisico (512 byte o 1K).
- `read` e `write` restituiscono il numero di byte letti o scritti, che può essere inferiore a quanto richiesto.
- Valori possibili di `whence`: `SEEK_SET` `SEEK_CUR` `SEEK_END`

```
/*****************************************
MODULO: lower.c
SCOPO: esempio di I/O non bufferizzato
*****************************************/
#include <stdio.h>
#include <ctype.h>
#include "mylib.h"
#define BUFLEN 1024
#define STDIN 0
#define STDOUT 1

void lowerbuf (char *s, int l)
{
  while (l-- > 0) {
    if (isupper(*s)) *s = tolower(*s);
    s++;
  }
}
```

```
int main (int argc, char *argv[])
{
  char buffer[BUFLEN];
  int x;

  while ((x=read(STDIN,buffer,BUFLEN)) > 0) {
    lowerbuf (buffer, x);
    x = write (STDOUT, buffer, x);
    if (x == -1)
      syserr (argv[0], "write() failure");
  }
  if (x != 0)
    syserr (argv[0], "read() failure");
  return 0;
}
```

Duplicazione di canali

```
int dup (int oldd)
```

- Duplica un file descriptor esistente e ne ritorna uno nuovo che ha in comune con il vecchio le seguenti proprietà:
  - si riferisce allo stesso file
  - ha lo stesso *puntatore* (per l'accesso casuale)
  - ha lo stesso modo di accesso.
- Proprietà importante: `dup` ritorna il primo descrittore libero a partire da 0!

Accesso ai direttori

- Sebbene sia possibile aprire e manipolare una directory con `open`, per motivi di portabilità è consigliato utilizzare le funzioni della libreria C (non system call)

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

DIR *opendir (char *dirname)
struct dirent *readdir (DIR *dirp)
void rewinddir (DIR *dirp)
int closedir (DIR *dirp)
```

- `opendir` apre la directory specificata (cfr. `fopen`)
- `readdir` ritorna un puntatore alla prossima entry della directory `dirp`
- `rewinddir` resetta la posizione del puntatore all'inizio
- `closedir` chiude la directory specificata

Accesso ai direttori

- Struttura interna di una directory:

```
struct dirent  {
        __ino_t   d_ino; /* inode # */
        __off_t   d_off;
        unsigned short int d_reclen; /* how large this structure really is */
        unsigned char d_type;
        char    d_name[256];
    };
```

- Campi della struttura `DIR`

```
typedef struct _dirdesc {
    int dd_fd;
    long    dd_loc;
    long    dd_size;
    long    dd_bbase;
    long    dd_entno;
    long    dd_bsize;
    char *  dd_buf;
} DIR;
```

```
/*****************************************
MODULO: dir.c
SCOPO: ricerca in un direttorio
*****************************************/
#include <string.h>
#include <sys/types.h>
#include <sys/dir.h>

int dirsearch( char*, char*, char*);

int main (int argc, char **argv)
{
   dirsearch (argv[1],argv[2],".");
}
```

```
int dirsearch (char *file1, char* file2,  char *dir)
{
  DIR *dp;
  struct direct *dentry;
  int status = 1;

  if ((dp=opendir (dir)) == NULL) return -1;
  for (dentry=readdir(dp); dentry!=NULL; dentry=readdir(dp))
  if ((strcmp(dentry->d_name,file1)==0)) {
    printf("Replacing entry %s with %s",dentry->d_name,file2);
    strcpy(dentry->d_name,file2);
    return 0;
  }
  closedir (dp);
  return status;
}
```

Accesso ai direttori

```
int chdir (char *dirname);
```

- Cambia la directory corrente e si sposta in *dirname*.
- E' necessario che la directory abbia il permesso di esecuzione

Gestione dei Link

```
#include <unistd.h>

int link (char *orig_name, char *new_name);
int unlink (char *file_name);
```

- `link` crea un link a `orig_name`. E' possibile fare riferimento al file con entrambi i nomi
- `unlink`
  - cancella un file cancellando l'$i\text{-}number$ nella directory entry
  - sottrae uno al link count nell'i-node corrispondente
  - se questo diventa zero, libera lo spazio associato al file
- `unlink` è l'unica system call per cancellare file !

```
#define TMP "/tmp"

int  fd;
char fname[32];
...
strcpy(fname,"myfile.xxx");
if ((fd = open(fname, O_WRONLY)) == -1) {
  perror(fname);
  return 1;
} else if (unlink(fname) == -1) {
  perror(fname);
  return 2;
} else {
  /* use temporary file */
}
...
```

Privilegi e accessi

```
#include <unistd.h>
int access (char *file_name, int access_mode);
```

- `access` verifica i permessi specificati in `access_mode` sul file `file_name`.
- I permessi sono una combinazione bitwise dei valori `R_OK`, `W_OK`, e `X_OK`.
- Specificando `F_OK` verifica se il file esiste
- Ritorna 0 se il file ha i permessi specificati

Privilegi e accessi

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (char *file_name, int mode);
int fchmod (int fildes, int mode);
```

- Permessi possibili: bitwise OR di

`S_ISUID`	04000	set user ID on execution
`S_ISGID`	02000	set group ID on execution
`S_ISVTX`	01000	save text image after execution
`S_IRUSR`	00400	read by owner
`S_IWUSR`	00200	write by owner
`S_IXUSR`	00100	execute (search on directory) by owner
`S_IRWXG`	00070	read, write, execute (search) by group
`S_IRWXO`	00007	read, write, execute (search) by others

Privilegi e accessi

```
#include <sys/types.h>
#include <sys/stat.h>
int chown (char *file_name, int owner, int group);
```

- `owner` = UID
- `group` = GID
- ottenibili con system call `getuid()` e `getgid()` (cfr. sezione sui processi)
- Solo super-user!

Stato di un file

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

int stat (char *file_name, struct stat *stat_buf);
int fstat (int fd, struct stat *stat_buf);
```

- Ritornano le informazioni contenute nell'i-node di un file
- L'informazione è ritornata dentro `stat_buf`.

Stato di un file

- Principali campi di `struct stat`:

```
dev_t         st_dev;      /* device */
ino_t         st_ino;      /* inode */
mode_t        st_mode;     /* protection */
nlink_t       st_nlink;    /* number of hard links */
uid_t         st_uid;      /* user ID of owner */
gid_t         st_gid;      /* group ID of owner */
dev_t         st_rdev;     /* device type (if inode device) */
off_t         st_size;     /* total size, in bytes */
unsigned long st_blksize;  /* blocksize for filesystem I/O */
unsigned long st_blocks;   /* number of blocks allocated */
time_t        st_atime;    /* time of last access */
time_t        st_mtime;    /* time of last modification */
time_t        st_ctime;    /* time of last change */
```

```
/* per stampare le informazioni con stat */
void display (char *fname, struct stat *sp)
{
  extern char *ctime();
  printf ("FILE %s\n", fname);
  printf ("Major number = %d\n", major(sp->st_dev));
  printf ("Minor number = %d\n", minor(sp->st_dev));
  printf ("File mode = %o\n", sp->mode);
  printf ("i-node number = %d\n", sp->ino);
  printf ("Links = %s\n", sp->nlink);
  printf ("Owner ID = %d\n", sp->st_uid);
  printf ("Group ID = %d\n", sp->st_gid);
  printf ("Size = %d\n", sp->size);
  printf ("Last access = %s\n", ctime(&sp->atime));
}
```

Controllo dei dispositivi

- Alcuni dispositivi (terminali, dispositivi di comunicazione) forniscono un insieme di comandi *device-specific*
- Questi comandi vengono eseguiti dai device driver
- Per questi dispositivi, il mezzo con cui i comandi vengono passati ai device driver è la system call `ioctl`.
- Tipicamente usata per determinare/cambiare lo stato di un terminale

```
#include <termio.h>
int ioctl(int fd, int request, structu termio *argptr);
```

- `request` è il comando device-specific, `argptr` definisce una struttura usata dal device driver eseguendo `request`.

Le variabili di ambiente

```
#include <stdlib.h>

char *getenv (char *env_var)
```

- Ritorna la definizione della variabile d'ambiente richiesta, oppure `NULL` se non è definita.
- E' possibile esaminare in sequenza tutte le variabili d'ambiente usando il terzo argomento del `main()`:

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

- Oppure accedendo la seguente variabile globale:

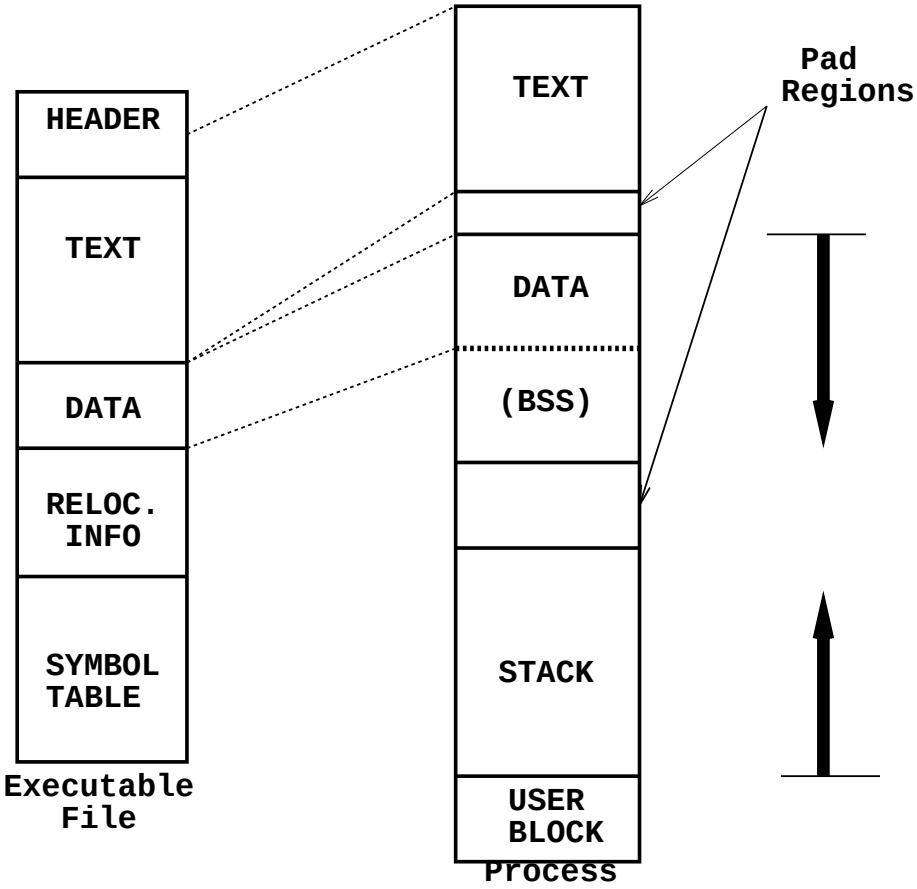
  ```
  extern char **environ;
  ```

```
/*****************************************
MODULO: env.c
SCOPO: elenco delle variabili d'ambiente
*****************************************/
#include <stdio.h>

int main (int argc, char *argv[], char *env[])
{
  puts ("Variabili d'ambiente:");
  while (*env != NULL)
    puts (*env++);
  return 0;
}
```

System Call per la Gestione dei Processi

Gestione dei processi

- Come trasforma UNIX un programma eseguibile in processo (con il comando `ld`)?



Gestione dei processi – Programma eseguibile

- HEADER: definita in `/usr/include/filehdr.h`.
  - definisce la dimensione delle altre parti
  - definisce l'entry point dell'esecuzione
  - contiene il *magic number*, numero speciale per la trasformazione in processo (system-dependent)
- TEXT: le istruzioni del programma
- DATA: I dati inizializzati (statici, extern)
- BSS (Block Started by Symbol): I dati non inizializzati (automatici). Nella trasformazione in processo, vengono messi tutti a zero in una sezione separata.
- RELOCATION: come il loader carica il programma. Rimosso dopo il caricamento
- SYMBOL TABLE: Può essere rimossa (`ld -s`) o con `strip` (toglie anche la relocation info).

Gestione dei processi – Processo

- TEXT: copia di quello del processo. Non cambia durante l'esecuzione
- DATA: possono crescere verso il basso ($heap$)
- BSS: occupa la parte bassa della sezione dati
- STACK: creato nella costruzione del processo. Contiene:
  - le variabili automatiche
  - i parametri delle procedure
  - gli argomenti del programma e le var. di ambiente
  - riallocato automaticamente dal sistema
  - cresce verso l'alto
- USER BLOCK (obsoleto): sottoinsieme delle informazioni mantenute dal sistema sul processo

Creazione di processi

```
#include <unistd.h>

pid_t fork (void)
```

- Crea un nuovo processo, figlio di quello corrente, che eredita dal padre:
  - I file aperti
  - Le variabili di ambiente
  - Tutti i settaggi dei segnali (v.dopo)
  - Directory di lavoro
- Al figlio viene ritornato `0`.
- Al padre viene ritornato il PID del figlio (o `-1` in caso di errore).
- NOTA: un processo solo chiama `fork`, ma è come se due processi ritornassero!

```
/*****************************************
MODULO: fork.c
SCOPO: esempio di creazione di un processo
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include "mylib.h"
int main (int argc, char *argv[])
{
   pid_t status;
   if ((status=fork()) == -1)
   syserr (argv[0], "fork() fallita");
   if (status == 0) {
      sleep(10);
      puts ("Io sono il figlio!");
   }
   else  sleep(2);
   printf ("Io sono il padre e mio figlio ha PID=%d)\n",status);
}
```

`fork` e debugging

- `gdb` non supporta automaticamente il debugging di programmi con `fork` $\Longrightarrow$ debugging sempre del padre
- Per debuggare il figlio:
  - Eseguire un `gdb` dello stesso programma da un'altra finestra
  - Usare il comando di `gdb`

    `attach` pid

    dove pid è il pid del figlio, determinato con `ps`
- Per garantire un minimo di sincronizzazione tra padre e figlio, è consigliato inserire una pausa condizionale all'ingresso del figlio

Esecuzione di un programma

```
#include <unistd.h>
int execl (char *file,char *arg0,char *arg1, ...,0)
int execlp(char *file,char *arg0,char *arg1, ...,0)
int execle(char *file,char *arg0,char *arg1, ...,0,char *envp[])
int execv (char *file,char *argv[])
int execvp(char *file,char *argv[])
int execve(char *file,char *argv[],char *envp[])
```

- Sostituiscono all'immagine attualmente in esecuzione quella specificata da `file`, che può essere:
  - un programma binario
  - un file di comandi
- In altri termini, `exec` trasforma un eseguibile in processo.
- NOTA: `exec` non ritorna!!

La Famiglia di `exec`

- `execl` utile quando so in anticipo il numero e gli argomenti, `execv` utile altrimenti.
- `execle` e `execve` ricevono anche come parametro la lista delle variabili d'ambiente.
- `execlp` e `execvp` utilizzano la variabile `PATH` per cercare il comando `file`.

```
/*****************************************
MODULO: exec.c
SCOPO: esempio d'uso di exec()
*****************************************/
#include <stdio.h>
#include <unistd.h>
#include "mylib.h"

int main (int argc, char *argv[])
{
  puts ("Elenco dei file in /tmp");
  execl ("/bin/ls", "ls", "/tmp", NULL);
  syserr (argv[0], "execl() fallita");
}
```

fork e exec

- Tipicamente `fork` viene usata con `exec`.
- Il processo figlio generato con `fork` viene usato per fare la `exec` di un certo programma.
- Esempio:

```
int pid = fork ();
if (pid == -1) {
  perror("");
} else if (pid == 0) {
  char *args [2];
  args [0] = "ls"; args [1] = NULL;
  execvp (args [0], args);
  exit (1);   /* vedi dopo */
} else {
  printf ("Sono il padre, e mio figlio e' %d.\n", pid);
}
```

Sincronizzazione tra padre e figli

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

void  exit(status)
void  _exit(status)
pid_t wait (int *status)
```

- `exit` è un wrapper all'effettiva system call `_exit()`
- `wait` sospende l'esecuzione di un processo fino a che uno dei figli termina.
  - Ne restituisce il PID ed il suo stato di terminazione, tipicamente ritornato come argomento dalla `exit`.
  - Restituisce `-1` se il processo non ha figli.
- Un figlio resta *zombie* da quando termina a quando il padre ne legge lo stato (con `wait()`).

Sincronizzazione tra padre e figli

- Lo stato può essere testato con le seguenti macro:

```
WIFEXITED(status)    WEXITSTATUS(status)    WIFSIGNALED(status)
WTERMSIG(status)     WIFSTOPPED(status)     WSTOPSIG(status)
```

- Informazione ritornata da `wait`
  - Se il figlio è terminato con `exit`
    - Byte 0: tutti zero
    - Byte 1: l'argomento della `exit`
  - Se il figlio è terminato con un segnale
    - Byte 0: il valore del segnale
    - Byte 1: tutti zero
- Comportamento di `wait` modificabile tramite segnali (v.dopo)

La Famiglia di `wait`

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

pid_t waitpid (pid_t pid, int *status, int options)
pid_t wait3 (int *status, int options, struct rusage *rusage)
```

- `waitpid` attende la terminazione di un particolare processo
  - `pid = -1`: tutti i figli
  - `pid = 0`: tutti i figli con stesso GID del processo chiamante
  - `pid` $<$ `-1` : tutti i figli con GID $= |$`pid`$|$
  - `pid > 0`: il processo `pid`
- `wait3` fornisce un'interfaccia alternativa per evitare l'attesa passiva del padre, e bloccare l'esecuzione solo in specifici casi `options`, scrivendo le relative informazioni in `rusage`.

```
/*****************************************
MODULO: wait.c
SCOPO: esempio d'uso di wait()
*****************************************/
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

#include "mylib.h"
int main (int argc, char *argv[])
{
  pid_t child;
  int status;
  if ((child=fork()) == 0) {
     sleep(5);
     puts ("figlio 1 - termino con stato 3");
     /*    _exit (3); */
  }
```

```
if (child == -1)
  syserr (argv[0], "fork() fallita");

if ((child=fork()) == 0) {
  puts ("figlio 2 - sono in loop infinito, uccidimi con:");
  printf ("  kill -9 %d\n", getpid());
  while (1) ;
}
if (child == -1)
  syserr (argv[0], "fork() fallita");


while ((child=wait(&status)) != -1) {
  printf ("il figlio con PID %d ", child);
  if (WIFEXITED(status)) {
    printf ("e' terminato (stato di uscita: %d)\n\n",
       WEXITSTATUS(status));
```

```
    } else if (WIFSIGNALED(status)) {
      printf ("e' stato ucciso (segnale omicida: %d)\n\n",
        WTERMSIG(status));
    } else if (WSTOPSIG(status)) {
      puts ("e' stato bloccato");
      printf ("(segnale bloccante: %d)\n\n",WSTOPSIG(status));
    } else
      puts ("non c'e' piu' !?");
  }
  return 0;
}
```

Informazioni sui processi

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

pid_t uid = getpid()
pid_t gid = getppid()
```

- `getpid` ritorna il PID del processo corrente
- `getppid` ritorna il PID del padre del processo corrente

```
/*****************************************
MODULO: fork2.c
SCOPO: funzionamento di getpid() e getppid()
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include "mylib.h"

int main (int argc, char *argv[])
{
  pid_t status;
  if ((status=fork()) == -1) {
    syserr (argv[0], "fork() fallita");
  }
  if (status == 0) {
    puts ("Io sono il figlio:\n");
    printf("PID = %d\tPPID = %d\n",getpid(),getppid());
  }
```

```
  else {
    printf ("Io sono il padre:\n");
    printf("PID = %d\tPPID = %d\n",getpid(),getppid());
  }
}
```

Informazioni sui processi – (cont.)

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

uid_t uid = getuid()
uid_t gid = getgid()
uid_t euid = geteuid()
uid_t egid = getegid()
```

- Ritornano la corrispondente informazione del processo corrente
- `geteuid` e `getegid` ritornano l'informazione sull'*effective* UID e GID, eventualmente settato con `chmod` (bit `s,S,t`).

Segnalazioni tra processi

- E' possibile spedire asincronamente dei segnali ai processi che hanno la nostra stessa UID:

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

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

- Valori possibili di `pid`:

  (`pid` > 0) segnale inviato al processo con PID=`pid`

  (`pid` $= 0$) segnale inviato a tutti i processi nel proprio gruppo

  (`pid` -1) segnale inviato a tutti i processi (tranne quelli di sistema)

  (`pid` < -1) segnale inviato a tutti i processi nel gruppo -`pid`

- *Gruppo di processi*: insieme dei processi aventi un antenato in comune.

Segnalazioni tra processi – (cont.)

- Il processo che riceve un segnale asincrono può specificare una routine da attivarsi alla sua ricezione.

```
#include <signal.h>

void (*signal (int sig, void (*func)(int)))
```

- `func` è la funzione da attivare, anche detta *signal handler*. Può essere una funzione definita dall'utente oppure:

  `SIG_DFL` per specificare il comportamento di default

  `SIG_IGN` per specificare di ignorare il segnale

- L'assegnazione di un handler rimane attiva fino a successiva `signal`.

Segnalazioni tra processi – (cont.)

- Segnali disponibilii (Linux): con il comando `kill -l`

SIGHUP	1+	Hangup	SIGINT	2+	Interrupt
SIGQUIT	3*	Quit	SIGILL	4*	Illegal instr.
SIGTRAP	5*	Trace trap	SIGABRT	6*	Abort signal.
SIGBUS	7*	Bus error	SIGFPE	8*	FP exception
SIGKILL	9+@	Kill	SIGUSR1	10+	User defined 1
SIGSEGV	11*	Segm. viol.	SIGUSR2	12+	User defined 2
SIGPIPE	13+	write on pipe	SIGALRM	14	Alarm clock
SIGTERM	15+	Software termination signal	-	16	
SIGCHLD	17#	Child stop/termination	SIGCONT	18	Continue after stop
SIGSTOP	19$@	Stop process	SIGTSTP	20$	Stop typed at tty
SIGTTIN	21$	Background read from tty	SIGTTOU	22$	Background write to tty
SIGURG	23#	Urgent condition on socket	SIGXCPU	24*	Cpu time limit
SIGXFSZ	25*	File size limit	SIGVTALRM	26+	Virtual time alarm
SIGPROF	27+	Profiling timer alarm	SIGWINCH	28#	Window size change
SIGIO	29+	I/O now possible	SIGPWR	30+	Power failure
SIGSYS	31+	Bad args to system call			

Segnalazioni tra processi – (cont.)

- Segnali con '+': azione di default = terminazione
- Segnali con '*': azione di default = terminazione e scrittura di un *core file*
- Segnali con '#': azione di default = ignorare il segnale
- Segnali con '$': azione di default = stoppare il processo
- Segnali con '@': non possono essere nè ignorati nè intercettati.
- I segnali 10 e 12 sono a disposizione dell'utente per gestire dei meccanismi di interrupt ad hoc.

  Sono tipicamente utilizzati insieme al *comando* `kill` per attivare la funzione desiderata in modo asincrono
- Esempio:
  Se un programma include l'istruzione `signal(SIGUSR1,int_proc);`, la funzione `int_proc` verrà eseguita tutte le volte che eseguo il comando

  `kill -10` *<PID del processo che esegue la* `signal`*>* .

```
/*****************************************
MODULO: signal.c
SCOPO: esempio di ricezione di segnali
*****************************************/
#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <signal.h>

long maxprim = 0;

void usr12_handler (int s)
{
  printf ("ricevuto segnale n.%d\n",s);
  printf ("il piu‘ grande primo trovato e‘ %ld\n",maxprim);
}
```

```
void good_bye (int s)
{
  printf ("il piu‘ grande primo trovato e‘ %ld\n",maxprim);
  printf ("ciao!\n");
  exit (1);
}

int is_prime (long x)
{
  long fatt,
maxfatt = (long)ceil(sqrt((double)x));
  if (x < 4) return 1;
  if (x % 2 == 0) return 0;

  for (fatt=3; fatt<=maxfatt; fatt+=2)
  return (x % fatt == 0 ? 0: 1);
}
```

```
int main (int argc, char *argv[])
{
  long n;
  int np=0;

  signal (SIGUSR1, usr12_handler);
  /* signal (SIGUSR2, usr12_handler); */
  signal (SIGHUP, good_bye);

  for (n=0; n<LONG_MAX; n++)
  if (is_prime(n)) {
    maxprim = n;
np++;
}
  printf ("%ld e' il piu' grande primo < %ld\n",LONG_MAX);
  printf ("%Totale dei  numeri primi=%d\n",np);
}
```

Segnali e terminazione di processi

- Il segnale `SIGCLD` viene inviato da un processo figlio che termina al padre
- L'azione di default è quella di ignorare il segnale (che causa la `wait()`a sbloccarsi)
- Può essere intercettato per modificare l'azione corrispondente
- (v. programma 7.18)

```
/*****************************************
MODULO: signal1.c
SCOPO: segnali e terminazione di processi
*****************************************/
#include <stdio.h>
#include <limits.h>
#include <math.h>
#include <signal.h>

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

  if (argc >= 1) {
     signal(SIGCHLD,SIG_IGN);
  }

  for (i=0; i<10; i++) {
```

```
      if (fork() == 0) {
    /* child i */
    printf("Child process #%d\n",i);
          exit(i);
}
  }
  retval = wait(&status);
  printf ("Wait: return value = %d\t return status = %d\n, \\
          retval, WEXITSTATUS(status));
}
```

Timeout e Sospensione

```
#include <unistd.h>

unsigned int alarm (unsigned seconds)
```

- `alarm` invia un segnale al processo chiamante dopo `seconds` secondi. Se `seconds` vale 0, l'allarme è annullato.
- Funzione di libreria C
- La chiamata resetta ogni precedente allarme
- Utile per implementare dei *timeout*, fondamentali per risorse utilizzate da più processi.
- Valore di ritorno:
  - 0 nel caso normale
  - Nel caso esistano delle `alarm()` con tempo residuo, il numero di secondi che mancavano all'allarme.

- Per cancellare eventuali allarmi sospesi: `alarm(0);`

Timeout e Sospensione

```
#include <unistd.h>

void pause ()
```

- Sospende un processo fino alla ricezione di un qualunque segnale.
- Ritorna sempre -1

System Call per la Comunicazione tra Processi (IPC)

Introduzione

- UNIX e IPC
- Pipe
- FIFO (*named pipe*)
- Code di messaggi (*message queue*)
- Memoria condivisa (*shared memory*)
- Semafori (cenni)

UNIX e IPC

- `ipcs`: riporta lo stato di tutte le risorse, o selettivamente, con le seguenti opzioni:
  - `-s` informazioni sui semafori;
  - `-m` informazioni sulle memorie condivise;
  - `-q` informazioni sulle code di messaggi.
- `ipcrm`: elimina le risorse (se permesso) dal sistema.
  - Nel caso di terminazioni anomale, le risorse possono rimanere allocate
  - Le opzioni sono quelle `ipcs`
  - Va specificato un ID di risorsa, come ritornato da `ipcs`

- Esempio:

```
host:user> ipcs
IPC status from /dev/kmem as of Wed Oct 16 12:32:13 1996
Message Queues:
T     ID     KEY        MODE       OWNER    GROUP
*** No message queues are currently defined ***

Shared Memory
T     ID     KEY        MODE       OWNER    GROUP
m   1300           0 D-rw-------     root   system
m   1301           0 D-rw-------     root   system
m   1302           0 D-rw-------     root   system

Semaphores
T     ID     KEY        MODE       OWNER    GROUP
*** No semaphores are currently defined ***
```

Pipe

- Il modo più semplice di stabilire un canale di comunicazione *unidirezionale e sequenziale* **in memoria** tra due processi consiste nel creare una *pipe*:

```
#include <unistd.h>

int pipe (int fildes[2])
```

- La chiamata ritorna zero in caso di successo, `-1` in caso di errore.
- Il primo descrittore (`[0]`) viene usato per leggere, il secondo `[1]` per scrivere.
- NOTA: L'interfaccia è quella dei file, quindi sono applicabili le system call che utilizzano file descriptor.

```
/*****************************************
MODULO: pipe.c
SCOPO: esempio di IPC mediante pipe
*****************************************/
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char *argv[])
{
  int status, p[2];
  char buf[64];

  pipe (p);
  if ((status=fork()) == -1)
    syserr (argv[0], "fork() fallita");
  else
```

```
  if (status == 0) { /* figlio */
     close (p[1]);
     if (read(p[0],buf,BUFSIZ) == -1)
       syserr (argv[0], "read() fallita");
     printf ("figlio - ricevuto: %s\n", buf);
     _exit (0);
  } else  {  /* padre */
     close (p[0]);
     write (p[1], "In bocca al lupo", 17);
     wait (&status);
     exit (0);
  }
}
```

Pipe e I/O

- Non è previsto l'accesso random (no `lseek`).
- La dimensione fisica delle pipe è limitata (dipendente dal sistema – BSD classico = 4K).
- L'operazione di `write` su una pipe è **atomica**
- La scrittura di un numero di Byte superiore a questo numero:
  - Blocca il processo scrivente fino a che non si libera spazio
  - la `write` viene eseguita a "pezzi", con risultati non prevedibili (es. più processi che scrivono)
- La `read` si blocca su pipe vuota e si sblocca non appena un Byte è disponibile (anche se ci sono meno dati di quelli attesi!)
- Chiusura prematura di un estremo della pipe:
  - scrittura: le `read` ritornano 0.
  - lettura: i processi in scrittura ricevono il segnale SIGPIPE (broken pipe)

Pipe e comandi

```
/*********************************************
MODULO: fork2.c
SCOPO: Realizzare il comando "ps | sort"
*********************************************/
#include <sys/types.h>
main ()
{
        pid_t pid;
        int pipefd[2];

        pipe (pipefd);
        if ((pid = fork()) == (pid_t)0) {
             close(1);       /* close stdout */
             dup (pipefd[1]);
             close (pipefd[0]);
             execlp ("ps", "ps", (char *)0);
```

```
        }
        else if (pid > (pid_t)0) {
             close(0); /* close stdin  */
             dup (pipefd[0]);
             close (pipefd[1]);
             execlp ("sort", "sort", (char *)0);
        }
}
```

Pipe e I/O non bloccante

- E' possibile forzare il comportamento di `write` e `read` rimuovendo la limitazione del bloccaggio.
- Utile per implementare meccanismi di polling su pipe
- Realizzato tipicamente con `fcntl` sul corrispondente file descriptor (0 o 1)

Pipe

- Limitazioni
  - possono essere stabilite soltanto tra processi *imparentati* (es., un processo ed un suo "progenitore", o tra due discendenti di un unico processo)
  - Non sono permanenti e sono distrutte quando il processo che le crea termina
- Soluzione: assegnare un nome *unico* alla pipe: *named pipe* dette anche FIFO.
- Funzionamento identico, ma il riferimento avviene attraverso il nome anzichè attraverso il file descriptor.
- Esistono fisicamente su disco e devono essere rimossi esplicitamente con `unlink`

Named Pipe (FIFO)

- Si creano con `mknod` (l'argomento `dev` viene ignorato)

  `mknod` (*nome*, `S_IFIFO|`*mode*, `0`)

- apertura, lettura/scrittura, chiusura avvengono come per un normale file
- Possono essere usate da processi non in relazione, in quanto il nome del file è unico nel sistema.
- Le operazioni di I/O su FIFO sono atomiche
- I/O normalmente bloccante, ma è possibile aprire (con `open`) un FIFO in modo non bloccante.

  In tal modo sia `read` che `write` saranno non bloccanti.

- Utilizzabile anche la funzione `mkfifo()`

```
/*****************************************
MODULO: fifo.c
SCOPO: esempio di IPC mediante named pipe
*****************************************/
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

int main (int argc, char *argv[])
{
  int i,fd;
  char buf[64];

  /* se argv[2] = NULL, legge dal FIFO, altrimenti scrive sul FIFO */
  mknod("/tmp/fifo",010777,0);


  if (argc == 2) {
```

```
    fd = open("/tmp/fifo",O_WRONLY);
  } else {
    fd = open("/tmp/fifo",O_RDONLY);
  }
  for (i=0;i<20;i++) {
    if (argc == 2) {
      write(fd,"HELLO",6);
    } else {
      read(fd,buf,6);
  printf("Ricevuto %s\n",buf);
    }
  }
}
```

Meccanismi di IPC Avanzati

- Cosiddette IPC SystemV:
  - Code di messaggi
  - Memoria condivisa
  - Semafori
- Disponibili API alternative (es. POSIX IPC).
  Particolarmente usate quelle per semafori!

Meccanismi di IPC Avanzati – (cont.)

- Caratteristiche comuni:
  - Una primitiva "`get`" per creare una nuova entry o recuperarne una esistente
  - Una primitiva "`ctl`" (control) per:
    * verificare lo stato di una entry,
    * cambiare lo stato di una entry
    * rimuovere una entry.

Meccanismi di IPC Avanzati – (cont.)

- La primitiva "`get`" specifica due informazioni associate ad ogni entry:
  - Una *chiave*, usata per la creazione dell'oggetto:
    - Valore intero arbitrario;
    - Valore intero generato con la funzione
      `key_t ftok(char *path, char id);`
      dato un nome di file esistente ed un carattere.
      Utile per evitare conflitti tra processi diversi;
    - `IPC_PRIVATE`, costante usata per creare una nuova entry
  - Dei *flag* di utilizzo:
    - `IPC_CREAT`: si crea una nuova entry se la chiave non esiste
    - `IPC_CREATE` + `IPC_EXCL`: si crea una nuova entry ad uso esclusivo da parte del processo
    - Permessi relativi all'accesso (tipo `rwxrwxrwx`)

Meccanismi di IPC Avanzati – (cont.)

- L'identificatore ritornato dalla "get" (se diverso da -1) è un descrittore utilizzabile dalle altre system call
- La creazione di un oggetto IPC causa anche l'inizializzazione di:
  - una struttura dati, diversa per i vari tipi di oggetto contenente informazioni su
    - UID, GID
    - PID dell'ultimo processo che l'ha modificata
    - Tempi dell'ultimo accesso o modifica
  - una struttura di permessi `ipc_perm`, contenente:

```
ushort cuid;    /* creator user id */
ushort cgid;    /* creator group id */
ushort uid; /* owner user id */
ushort gid; /* owner group id */
ushort mode; /* r/w permissions */
```

Code di Messaggi

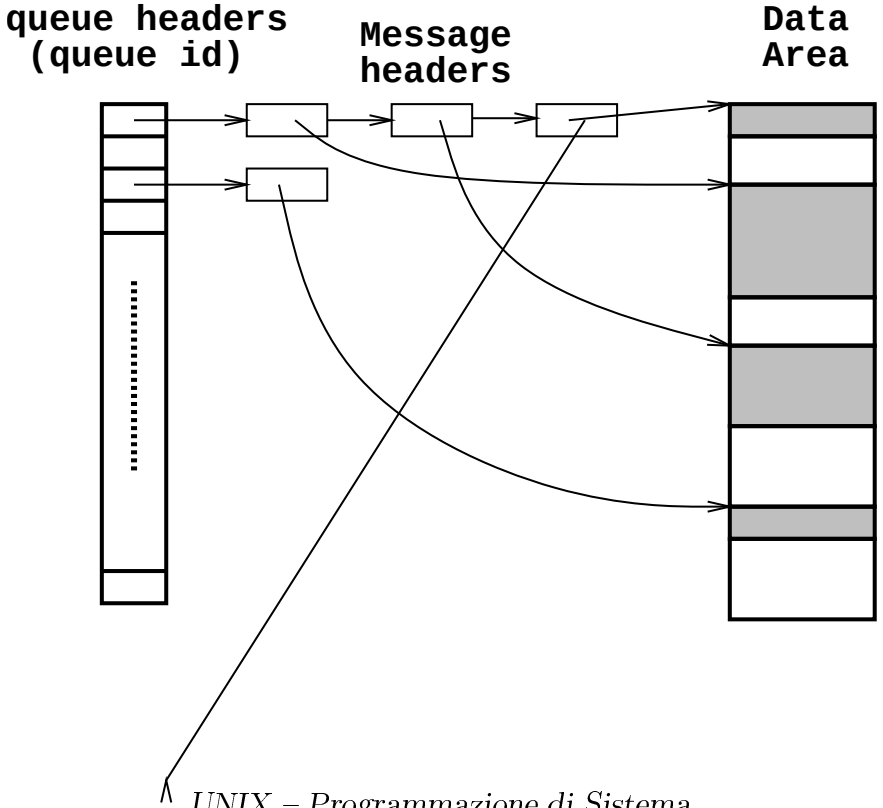
- Un messaggio è una unità di informazione di dimensione variabile, senza un formato predefinito
- Vengono memorizzati nelle *code*, che vengono individuate dalla chiave

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

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

- Crea una coda di messaggi data la chiave `key` (di tipo `long`) se
  - `key` = `IPC_PRIVATE`, oppure
  - `key` non è definita, e `flag & IPC_CREAT` è vero.

Code di Messaggi – (cont.)

- I permessi associati ad una entry vengono specificati nei 9 LSb del campo `flag` (cfr. `creat`).
- Hanno significato solo i flag di lettura e scrittura.
- Struttura delle code di messaggi:



Code di Messaggi: Gestione

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

int msgctl (int id, int command, struct msqid_ds *buffer)
```

- `id` è il descrittore ritornato da `msgget`
- `command`:
  - `IPC_RMID` Cancella la coda (`buffer` non usato)
  - `IPC_STAT` Ritorna informazioni relative alla coda nella struttura puntata da `buffer` (contiene info su UID, GID, stato della coda)
  - `IPC_SET` Modifica un sottoinsieme dei campi contenuti nella struct

Code di Messaggi: Gestione – (Cont.)

- `buffer` è un puntatore a una struttura definita in `sys/msg.h` contenente (campi utili):

```
struct msqid_ds
{
  struct ipc_perm msg_perm;     /* permissions (rwxrwxrwx) */
  __time_t msg_stime;           /* time of last msgsnd command */
  __time_t msg_rtime;           /* time of last msgrcv command */
  __time_t msg_ctime;           /* time of last change */
  unsigned long int __msg_cbytes; /* current number of bytes on queue */
  msgqnum_t msg_qnum;           /* number of messages currently on queue */
  msglen_t msg_qbytes;          /* max number of bytes allowed on queue */
  __pid_t msg_lspid;            /* pid of last msgsnd() */
  __pid_t msg_lrpid;            /* pid of last msgrcv() */
};
```

```
/*****************************************
MODULO: msgctl.c
SCOPO: Illustrare il funz. di msgctl()
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <time.h>
void do_msgctl();
char warning_message[] = "If you remove read permission\\
  for yourself, this program will fail frequently!";
main()
{
struct msqid_ds buf;
int cmd, /* command to be given to msgctl() */
msqid; /* queue ID to be given to msgctl() */
 fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
```

```
 fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
 fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
 fprintf(stderr, "\nEnter the value for the command: ");
 scanf("%i", &cmd);
 switch (cmd) {
 case IPC_SET:
  fprintf(stderr, "Before IPC_SET, get current values:");
  /* fall through to IPC_STAT processing */
  case IPC_STAT:
do_msgctl(msqid, IPC_STAT, &buf);
      fprintf(stderr, "msg_perm.uid = %d\n", buf.msg_perm.uid);
      fprintf(stderr,"msg_perm.gid = %d\n", buf.msg_perm.gid);
      fprintf(stderr, "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
      fprintf(stderr, "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
      fprintf(stderr, "msg_perm.mode = %#o, ", buf.msg_perm.mode);
      fprintf(stderr, "access permissions = %#o\n", buf.msg_perm.mode &
      fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
      fprintf(stderr, "msg_qbytes = %d\n",buf.msg_qbytes);
```

```
fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
if (buf.msg_time) {
fprintf(stderr, "msg_stime = %s\n", ctime(&buf.msg_stime));
}
if (buf.msg_rtime) {
fprintf(stderr, "msg_rtime = %s\n", ctime(&buf.msg_rtime));
fprintf(stderr, "msg_ctime = %s", ctime(&buf.msg_ctime));
if (cmd == IPC_STAT)
    break;
/* Now continue with IPC_SET. */
fprintf(stderr, "Enter msg_perm.uid: ");
scanf ("%hi", &buf.msg_perm.uid);
fprintf(stderr, "Enter msg_perm.gid: ");
scanf("%hi", &buf.msg_perm.gid);
fprintf(stderr, "%s\n", warning_message);
fprintf(stderr, "Enter msg_perm.mode: ");
```

```
        scanf("%hi", &buf.msg_perm.mode);
        fprintf(stderr, "Enter msg_qbytes: ");
        scanf("%hi", &buf.msg_qbytes);
       do_msgctl(msqid, IPC_SET, &buf);
       break;
  case IPC_RMID:
  default:
       /* Remove the message queue or try an unknown command. */
       do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
       break;
   }
   _exit(0);
}

void do_msgctl(int msqid, int cmd, struct msqid_ds* buf)
{
int rtrn; /* hold area for return value from msgctl() */
fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
```

```
msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
rtrn = msgctl(msqid, cmd, buf);
if (rtrn == -1) {
perror("msgctl: msgctl failed");
_exit(1);
} else {
 fprintf(stderr, "msgctl: msgctl returned %d\n", rtrn);
}
}
```

Code di Messaggi: Scambio di Informazione

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int id,struct msgbuf *msg,size_t size,int flag)
int msgrcv(int id,struct msgbuf *msg,size_t size,long type,int flag);
```

- `id` è il descrittore ritornato da `msgget`
- Struttura dei messaggi:

```
struct msgbuf {
   long    mtype;      /* message type */
   char    mtext[1];   /* message text */
};
```

- Da interpretare come "template" di messaggio!
- In pratica, si usa una `struct` costruita dall'utente

Code di Messaggi: Scambio di Informazione – (cont.)

- `flag`:

`IPC_NOWAIT`	(`msgsnd` e `msgrcv`)	non si blocca se non ci sono messaggi da leggere
`MSG_NOERRROR`	(`msgrcv`)	tronca i messaggi a `size` byte senza errore

- `type` indica quale messaggio prelevare:
  - 0 Il primo messaggio, indipendentemente dal tipo
  - > 0 Il primo messaggio di tipo `type`
  - < 0 Il primo messaggio con tipo più "vicino" al valore assoluto di `type`

```
/*************************************
  SERVER process
*************************************/
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY  75
#define MSGTYPE  1

main (int argc,char **argv)
{
key_t msgkey;
int msgid, pid;
```

```
    struct msg {
int mtype;
char mtext[256];
} Message;

if ((msgid = msgget(MSGKEY,(0666|IPC_CREAT|IPC_EXCL))) == -1) {
       perror(argv[0]);
    }
    /* leggo dalla coda */
    msgrcv(msgid,&Message,sizeof(Message.mtext),MSGTYPE,0);  /* WAIT */
    printf("Received from client: %s\n",Message.mtext);
    pid = getpid();
    sprintf(Message.mtext,"%d",pid);
    Message.mtype = MSGTYPE;
    msgsnd(msgid,&Message,sizeof(Message.mtext),0);  /* WAIT */
}
```

```
/*************************************
  CLIENT process
*************************************/
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSGKEY  75
#define MSGTYPE  1

main (int argc,char **argv)
{
key_t msgkey;
int msgid, pid;
```

```
struct msg {
int mtype;
char mtext[256];
} Message;

if ((msgid = msgget(MSGKEY,0666)) == -1) {
       syserr(argv[0],"");
    }
    /* scrivo il PID nella coda */
    pid = getpid();
    sprintf(Message.mtext,"%d",pid);
    Message.mtype = MSGTYPE;
    msgsnd(msgid,&Message,sizeof(Message.mtext),0);  /* WAIT */
    msgrcv(msgid,&Message,sizeof(Message.mtext),MSGTYPE,0);  /* WAIT */
    printf("Received message from server: %s\n",Message.mtext);
}
```

Memoria Condivisa

- Due o più processi possono comunicare anche condividendo una parte del loro spazio di indirizzamento (virtuale).
- Questo spazio condiviso è detto *memoria condivisa* (*shared memory*), e la comunicazione avviene scrivendo e leggendo questa parte di memoria

```
#include <sys/shm.h>
#include <sys/ipc.h>

shm_id  shmget(key_t key, int size, int flags);
```

- I parametri hanno lo stesso significato di quelli utilizzati da `msgget`.
- `size` indica la dimensione in byte della regione condivisa.

Memoria Condivisa – (cont.)

- Una volta creata, l'area di memoria non è subito disponibile
- Deve essere *collegata* all'area dati dei processi che vogliono utilizzarla.

```
#include <sys/shm.h>
#include <sys/ipc.h>

char *shmat (int shmid, char *shmaddr, int flag)
```

- `shmaddr` indica l'indirizzo virtuale dove il processo vuole `attaccare` il segmento di memoria condivisa.
- Il valore di ritorno rappresenta l'indirizzo di memoria condivisa effettivamente risultante

Memoria Condivisa – (cont.)

- In base ai valori di `flag` e di `shmaddr` si determina il punto di attacco del segmento:

`shmaddr = 0 && (flag & SHM_RND)`	al primo indirizzo disponibile
`shmaddr != 0 && !(flag & SHM_RND)`	all'indirizzo indicato da `shmaddr`

- Il segmento è attaccato in lettura se `flag & SHM_RDONLY` è vero, contemporaneamente in lettura e scrittura altrimenti.
- Un segmento attaccato in precedenza può essere "staccato" (*detached*) con `shmdt`

```
int shmdt (char *shmaddr)
```

  dove `shmaddr` è l'indirizzo che individua il segmento di memoria condivisa.

- Non viene passato l'ID della regione perchè è possibile avere più aree di memoria identificate dallo stesso ID (cioè attaccate ad indirizzi diversi);

Memoria Condivisa: Gestione

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

int shmctl (int shmid, int cmd, struct shmid_ds *buffer);
```

- `shmid` è il descrittore ritornato da `shmget`
- Valori di `cmd`:

`IPC_RMID`	Cancella il segm. di memoria condivisa
`IPC_STAT`	Ritorna informazioni relative alla coda nella struttura puntata da `buffer` (contiene info su UID, GID, permessi, stato della coda)
`IPC_SET`	Modifica un sottoinsieme dei campi contenuti nella struct (UID, GID, permessi)
`SHM_LOCK`	Impedisce che il segmento venga swappato o paginato

Memoria Condivisa: Gestione – (Cont.)

- `buffer` è un puntatore a una struttura definita in `sys/shm.h` contenente:

```
struct shmid_ds   {
    struct ipc_perm shm_perm;          /* operation permission struct */
    size_t shm_segsz;                  /* size of segment in bytes */
    __time_t shm_atime;                /* time of last shmat() */
    __time_t shm_dtime;                /* time of last shmdt() */
    __time_t shm_ctime;                /* time of last change by shmctl() */
    __pid_t shm_cpid;                  /* pid of creator */
    __pid_t shm_lpid;                  /* pid of last shmop */
    shmatt_t shm_nattch;               /* number of current attaches */
  };
```

```
/*****************************************
MODULO: msgctl.c
SCOPO: Illustrare il funz. di msgctl()
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <time.h>
void do_shmctl();

int main()
{
     int cmd; /* command code for shmctl() */
     int shmid; /* segment ID */
     struct shmid_ds shmid_ds;
     fprintf(stderr, "Enter the shmid for the desired segment: ");
     scanf("%i", &shmid);
     fprintf(stderr, "Valid shmctl cmd values are:\n");
     fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
```

```
fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
fprintf(stderr, "Enter the desired cmd value: ");
scanf("%i", &cmd);
switch (cmd) {
case IPC_STAT:
  /* Get shared memory segment status. */
  break;
case IPC_SET:
  do_shmctl(shmid, IPC_STAT, &shmid_ds);
  /* Set UID, GID, and permissions to be loaded. */
  fprintf(stderr, "\nEnter shm_perm.uid: ");
  scanf("%hi", &shmid_ds.shm_perm.uid);
  fprintf(stderr, "Enter shm_perm.gid: ");
  scanf("%hi", &shmid_ds.shm_perm.gid);
  fprintf(stderr, "Note: Keep read permission for yourself.\n");
  fprintf(stderr, "Enter shm_perm.mode: ");
  scanf("%hi", &shmid_ds.shm_perm.mode);
```

```
      break;
     case IPC_RMID: /* Remove the segment */
      break;
     case SHM_LOCK: /* Lock the shared memory segment. */
      break;
     case SHM_UNLOCK: /* Unlock the shared memory segment. */
      break;
     default: /* Unknown command will be passed to shmctl. */
     break;
    }
    do_shmctl(shmid, cmd, &shmid_ds);
    _exit(0);
}

void do_shmctl(int shmid, int cmd, struct shmid_ds* buf)
{
    int rtrn; /* hold area */
    fprintf(stderr, "shmctl: Calling shmctl(%d, %d, buf)\n", shmid, cmd);
    if (cmd == IPC_SET) {
     fprintf(stderr, "\tbuf->shm_perm.uid == %d\n", buf->shm_perm.uid);
```

```
 fprintf(stderr, "\tbuf->shm_perm.gid == %d\n", buf->shm_perm.gid);
 fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n", buf->shm_perm.mode);
}
if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
    perror("shmctl: shmctl failed");
    _exit(1);
} else {
     fprintf(stderr, "shmctl: shmctl returned %d\n", rtrn);
}
if (cmd != IPC_STAT && cmd != IPC_SET)
    return;
/* Print the current status. */
 fprintf(stderr, "\nCurrent status:\n");
 fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
 fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
 fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
 fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
 fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
 fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.key);
 fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
```

```
    fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
    fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
    fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
    if (buf->shm_atime)
       fprintf(stderr, "\tshm_atime = %s", ctime(&buf->shm_atime));
    if (buf->shm_dtime)
       fprintf(stderr, "\tshm_dtime = %s", ctime(&buf->shm_dtime));
    fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
}
```

```
/*************************************************************
NOME: shm1.c
SCOPO: ‘‘attaccare’’ due volte un area di memoria condivisa
*************************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define K 1
#define SHMKEY 75
#define N 20

int shmid;
int main (int argc, char **argv) {
  int i, *pint;
  char *addr1, *addr2;
  shmid = shmget(SHMKEY, 128*K, 0777|IPC_CREAT);
  addr1 = shmat(shmid,0,0);
  addr2 = shmat(shmid,0,0);
```

```
  printf("Address1 = 0x%x\t Address2 = 0x%x\t\n", addr1,addr2);
  /* scrivi nella regione 1 */
  pint = (int*)addr1;
  for (i=0;i<N;i++) {
    *pint = i;
    printf("Writing: Index %4d\tValue: %4d\n",i,*pint++);
  }
  /* leggi dalla regione 2 */
  pint = (int*)addr2;
  for (i=0;i<N;i++) {
    printf("Reading: Index %4d\tValue: %4d\n",i,*pint++);
  }
}
```

```
/*****************************************************************
NOME: shm2.c
SCOPO: ‘‘attaccarsi’’ ad un area di memoria condivisa
*****************************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define K 1
#define N 20
#define SHMKEY 75

int shmid;
int main (int argc, char **argv) {
  int i, *pint;
  char *addr;
  shmid = shmget(SHMKEY, 128*K, 0777);
  addr = shmat(shmid,0,0);
```

```
  printf("Address = 0x%x\n", addr);
  pint = (int*) addr;
/* leggi dalla regione attaccata in precedenza */
  for (i=0;i<N;i++) {
    printf("Reading: (Value = %4d)\n",*pint++);
  }
}
```

```
/*****************************************
MODULO: shm_server.c
SCOPO: server memoria condivisa
*****************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27
int main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5678;
```

```
    /* Create the segment */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        _exit(1);
    }
    /* Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        _exit(1);
    }
    s = shm;
    for (c = 'a'; c <= 'z'; c++)    *s++ = c;
    *s = NULL;
    while (*shm != '*')
       sleep(1);
    printf("Received '*'. Exiting...\n");
    _exit(0);
}
```

```
/*****************************************
MODULO: shm_client.c
SCOPO: client memoria condivisa
*****************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ 27
int main()
{
    int shmid;
    key_t key;
    char *shm, *s;
    key = 5678;
    /* Locate the segment */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
```

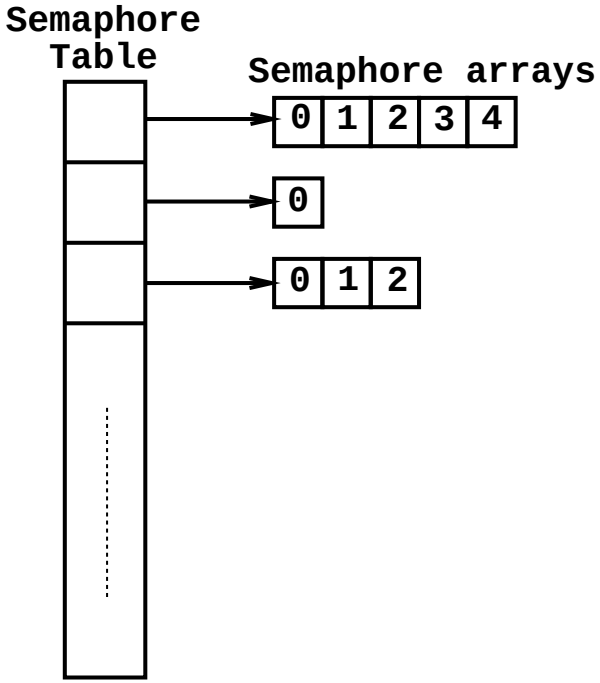
```
        _exit(1);
    }

    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
    exit(1);
    }

    printf("Content of shared memory segment: ");
    for (s = shm; *s != NULL; s++)    putchar(*s);
    putchar('\n');

    sleep(3);
    *shm = '*';
    _exit(0);
}
```

Sincronizzazione tra Processi

- I semafori permettono la sincronizzazione dell'esecuzione di due o più processi
  - Sincronizzazione su un dato valore
  - Mutua esclusione
- Semafori SystemV:
  - piuttosto diversi da semafori classici
  - "pesanti" dal punto di vista della gestione
- Disponibili varie API (per es. POSIX semaphores)

Semafori (SystemV API)

- Non è possibile allocare un singolo semaforo, ma è necessario crearne un insieme (vettore di semafori)
- Struttura interna di un semaforo



Semafori (SystemV API)

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

int semid = semget(int key, int count, short flags);
```

- I valori di `key` e di `flags` sono identici al caso delle code di messaggi e della shared memory.
- `count` è il numero di semafori identificati dal `semid` (quanti semafori sono contenuti nel vettore).
- NOTA: I semafori hanno sempre valore iniziale $= 0 \Rightarrow$ potenziali deadlock durante la creazione!

Operazioni su Semafori

```
int semctl(int semid, int semnum, int command, union semun *args);
union semun {
   int val;
   struct semid* buffer;
   unsigned short *array;
};
```

- Operazioni (`command`):

`IPC_RMID`*	Rimuove il set di semafori
`IPC_SET`@	Modifica il set di semafori
`IPC_STAT`@	Statistiche sul set di semafori
`GETVAL`*	legge il valore del semaforo **`semnum`** in **`args.val`**
`GETALL`#	legge tutti i valori in **`args.array`**
`SETVAL`*	assegna il valore del semaforo **`semnum`** in **`args.val`**
`SETALL`#	assegna tutti i valori con i valori in **`args.array`**
`GETPID`*	Valore di PID dell'ultimo processo che ha fatto operazioni
`GETNCNT`*	numero di processi in attesa che un semaforo aumenti
`GETZCNT`*	numero di processi in attesa che un semaforo diventi 0

Operazioni su Semafori

- Operazioni con "*": `arg` = `arg.val` (intero)
- Operazioni con "#": `arg` = `arg.array` (puntatore a array di `short`)
- Operazioni con "@": `arg` = `arg.buf` (puntatore a buffer di tipo `semid_ds`)
- `buffer` è un puntatore ad una struttura `semid_ds` definita in `sys/sem.h`:

```
struct semid_ds   {
    struct ipc_perm shm_perm;           /* operation permission struct */
  };
```

```
/*****************************************
MODULO: semctl.c
SCOPO: Illustrare il funz. di semctl()
*****************************************/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>
#include    <time.h>

struct semid_ds semid_ds;

/* explicit declaration required */
union semun {
  int val;
  struct semid_ds *buf;
  u_short *array;
} arg;
```

```
#ifdef LINUX
union semun {
    int val;
    struct semid_ds* buf;
    unsigned short int *array;
    struct seminfo *__buf;
} arg;
#endif

void do_semctl(int,int,int,union semun);
void do_stat(void);
char    warning_message[] = "If you remove read permission\
        for yourself, this program will fail frequently!";

int main()
{
   union semun    arg;    /* union to pass to semctl() */
   int    cmd,    /* command to give to semctl() */
            i,    /* work area */
        semid,    /* semid to pass to semctl() */
```

```
    semnum;    /* semnum to pass to semctl() */


fprintf(stderr, "Enter semid value: ");
scanf("%i", &semid);

fprintf(stderr, "Valid semctl cmd values are:\n");
fprintf(stderr, "\tGETALL = %d\n", GETALL);
fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
fprintf(stderr, "\tGETPID = %d\n", GETPID);
fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
fprintf(stderr, "\tSETALL = %d\n", SETALL);
fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
fprintf(stderr, "\nEnter cmd: ");
scanf("%i", &cmd);
```

```
 /* Do some setup operations needed by multiple commands. */
 switch (cmd) {
  case GETVAL:
  case SETVAL:
  case GETNCNT:
  case GETZCNT:
  /* Get the semaphore number for these commands. */
    fprintf(stderr, "\nEnter semnum value: ");
    scanf("%i", &semnum);
    break;
case GETALL:
case SETALL:
  /* Allocate a buffer for the semaphore values. */
    fprintf(stderr,"Get number of semaphores in the set.\n");
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    if (arg.array =(u_short *)malloc((unsigned)
        (semid_ds.sem_nsems * sizeof(u_short)))) {
       /* Break out if you got what you needed. */
       break;
```

```
    }
    fprintf(stderr,"semctl: unable to allocate space for %d values\n",
            semid_ds.sem_nsems);
    _exit(2);
}



/* Get the rest of the arguments needed for the specified command. */
switch (cmd) {
case SETVAL:
    /* Set value of one semaphore. */
    fprintf(stderr, "\nEnter semaphore value: ");
    scanf("%i", &arg.val);
    do_semctl(semid, semnum, SETVAL, arg);
    /* Fall through to verify the result. */
    fprintf(stderr, "Do semctl GETVAL command to verify results.\n");
case GETVAL:
   /* Get value of one semaphore. */
   arg.val = 0;
```

```
    do_semctl(semid, semnum, GETVAL, arg);
    break;
  case GETPID:
    /* Get PID of last process to successfully complete a
     semctl(SETVAL), semctl(SETALL), or semop() on the
     semaphore. */
    arg.val = 0;
    do_semctl(semid, 0, GETPID, arg);
    break;
   case GETNCNT:
    /* Get number of processes waiting for semaphore value to increase. */
    arg.val = 0;
    do_semctl(semid, semnum, GETNCNT, arg);
    break;
   case GETZCNT:
    /* Get number of processes waiting for semaphore value to become zero. */
    arg.val = 0;
    do_semctl(semid, semnum, GETZCNT, arg);
    break;
   case SETALL:
```

```
 /* Set the values of all semaphores in the set. */
  fprintf(stderr, "There are %d semaphores in the set.\n", semid_ds.sem_nsems);
  fprintf(stderr, "Enter semaphore values:\n");
  for (i = 0; i < semid_ds.sem_nsems; i++) {
    fprintf(stderr, "Semaphore %d: ", i);
    scanf("%hi", &arg.array[i]);
  }
  do_semctl(semid, 0, SETALL, arg);
  /* Fall through to verify the results. */
  fprintf(stderr, "Do semctl GETALL command to verify results.\n");
case GETALL:
  /* Get and print the values of all semaphores in the set.*/
  do_semctl(semid, 0, GETALL, arg);
  fprintf(stderr, "The values of the %d semaphores are:\n", semid_ds.sem_nsems);
  for (i = 0; i < semid_ds.sem_nsems; i++)
  fprintf(stderr, "%d ", arg.array[i]);
  fprintf(stderr, "\n");
  break;
case IPC_SET:
  /* Modify mode and/or ownership. */
```

```
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    fprintf(stderr, "Status before IPC_SET:\n");
    do_stat();
    fprintf(stderr, "Enter sem_perm.uid value: ");
    scanf("%hi", &semid_ds.sem_perm.uid);
    fprintf(stderr, "Enter sem_perm.gid value: ");
    scanf("%hi", &semid_ds.sem_perm.gid);
    fprintf(stderr, "%s\n", warning_message);
    fprintf(stderr, "Enter sem_perm.mode value: ");
    scanf("%hi", &semid_ds.sem_perm.mode);
    do_semctl(semid, 0, IPC_SET, arg);
    /* Fall through to verify changes. */
    fprintf(stderr, "Status after IPC_SET:\n");
  case IPC_STAT:
    /* Get and print current status. */
    arg.buf = &semid_ds;
    do_semctl(semid, 0, IPC_STAT, arg);
    do_stat();
    break;
```

```
    case IPC_RMID:
      /* Remove the semaphore set. */
      arg.val = 0;
      do_semctl(semid, 0, IPC_RMID, arg);
      break;
    default:
      /* Pass unknown command to semctl. */
      arg.val = 0;
      do_semctl(semid, 0, cmd, arg);
      break;
   }
 exit(0);
}

void do_semctl(int semid, int semnum, int cmd, union semun arg)
{
   register int      i;   /* work area */

   fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ", semid, semnum, cmd);
   switch (cmd) {
```

```
 case GETALL:
   fprintf(stderr, "arg.array = %#x)\n", arg.array);
   break;
  case IPC_STAT:
  case IPC_SET:
   fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
   break;
  case SETALL:
   fprintf(stderr, "arg.array = [", arg.buf);
   for (i = 0;i < semid_ds.sem_nsems;) {
     fprintf(stderr, "%d", arg.array[i++]);
    if (i < semid_ds.sem_nsems)
       fprintf(stderr, ", ");
   }
   fprintf(stderr, "])\n");
   break;
  case SETVAL:
  default:
    fprintf(stderr, "arg.val = %d)\n", arg.val);
break;
```

```
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == -1) {
  perror("semctl: semctl failed");
  exit(1);
 }
  fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}

void do_stat()
{
  fprintf(stderr, "sem_perm.uid = %d\n",
       semid_ds.sem_perm.uid);
  fprintf(stderr, "sem_perm.gid = %d\n",
       semid_ds.sem_perm.gid);
  fprintf(stderr, "sem_perm.cuid = %d\n",
       semid_ds.sem_perm.cuid);
  fprintf(stderr, "sem_perm.cgid = %d\n",
       semid_ds.sem_perm.cgid);
```

```
  fprintf(stderr, "sem_perm.mode = %#o, ",
       semid_ds.sem_perm.mode);
  fprintf(stderr, "access permissions = %#o\n",
       semid_ds.sem_perm.mode & 0777);
  fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
  fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
      ctime(&semid_ds.sem_otime) : "Not Set\n");
  fprintf(stderr, "sem_ctime = %s",
       ctime(&semid_ds.sem_ctime));
}
```

Operazioni su Semafori

```
int oldval = semop(int id, struct sembuf* ops, int count);
```

- Applica l'insieme `ops` di operazioni (in numero pari a `count`) al semaforo `id`.
- Le operazioni, contenute in un vettore opportunamente allocato, sono descritte dalla `struct sembuf`:

```
struct sembuf {
  short sem_num;
  short sem_op;
  short sem_flg;
};
```

  - `sem_num`: semaforo su cui l'operazione (i-esima) viene applicata
  - `sem_op`: l'operazione da applicare
  - `sem_flag`: le modalità con cui l'operazione viene applicata

Operazioni su Semafori

- Valori di `sem_op`:

< 0	equivale a `P` (si blocca se `sem_val` $\leq$ `0`) Decrementa il semaforo della quantità `ops.sem_op`
$= 0$	In attesa che il valore del semaforo diventi 0
> 0	equivalente a `V` Incrementa il semaforo della quantità `ops.sem_op`

- Valori di `sem_flg`:

`IPC_NOWAIT`	Per realizzare `P` e `V` non bloccanti (comodo per realizzare *polling*)
`SEM_UNDO`	Ripristina il vecchio valore quando termina (serve nel caso di terminazioni precoci)

```
/*****************************************
MODULO: semop.c
SCOPO: Illustrare il funz. di semop()
*****************************************/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

int ask(int*,struct sembuf **);

static struct semid_ds  semid_ds;           /* status of semaphore set */
static char    error_mesg1[] = "semop: Can't allocate space for %d\
        semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for %d\
        sembuf structures. Giving up.\n";

int main()
{
```

```
register int    i;   /* work area */
int    nsops;   /* number of operations to do */
int    semid;   /* semid of semaphore set */
struct sembuf    *sops;   /* ptr to operations to perform */

/* Loop until the invoker doesn't want to do anymore. */
while (nsops = ask(&semid, &sops)) {
 /* Initialize the array of operations to be performed.*/
 for (i = 0; i < nsops; i++) {
   fprintf(stderr,"\nEnter values for operation %d of %d.\n",i+1,nsops);
   fprintf(stderr,"sem_num(valid values are 0 <= sem_num < %d): ",
           semid_ds.sem_nsems);
   scanf("%hi", &sops[i].sem_num);
   fprintf(stderr, "sem_op: ");
   scanf("%hi", &sops[i].sem_op);
   fprintf(stderr,"Expected flags in sem_flg are:\n");
   fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",IPC_NOWAIT);
   fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",SEM_UNDO);
   fprintf(stderr, "sem_flg: ");
   scanf("%hi", &sops[i].sem_flg);
```

```
  }

  /* Recap the call to be made. */
  fprintf(stderr,"\nsemop: Calling semop(%d, &sops, %d) with:",semid, nsops);
  for (i = 0; i < nsops; i++) {
    fprintf(stderr, "\nsops[%d].sem_num = %d, ",i,sops[i].sem_num);
    fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
    fprintf(stderr, "sem_flg = %#o\n",sops[i].sem_flg);
  }

  /* Make the semop() call and report the results. */
  if ((i = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
  } else {
    fprintf(stderr, "semop: semop returned %d\n", i);
  }
 }
}
```

```
int ask(semidp, sopsp)
{
 static union semun     arg;  /* argument to semctl */
 int     i;  /* work area */
 static int     nsops = 0;  /* size of currently allocated sembuf array */
 static int     semid = -1;   /* semid supplied by user */
 static struct sembuf    *sops;     /* pointer to allocated array */

 if (semid < 0) {
  /* First call; get semid from user and the current state of
     the semaphore set. */
  fprintf(stderr,"Enter semid of the semaphore set you want to use: ");
  scanf("%i", &semid);
  *semidp = semid;
  arg.buf = &semid_ds;
  if (semctl(semid, 0, IPC_STAT, arg) == -1) {
    perror("semop: semctl(IPC_STAT) failed");
    /* Note that if semctl fails, semid_ds remains filled with zeros,
       so later test for number of semaphores will be zero. */
```

```
    fprintf(stderr,"Before and after values are not printed.\n");
  } else if ((arg.array = (ushort *)malloc((unsigned)
             (sizeof(ushort) * semid_ds.sem_nsems))) == NULL) {
     fprintf(stderr, error_mesg1,semid_ds.sem_nsems);
    exit(1);
  }
}

/* Print current semaphore values. */
if (semid_ds.sem_nsems) {
  fprintf(stderr, "There are %d semaphores in the set.\n", semid_ds.sem_nsems);
  if (semctl(semid, 0, GETALL, arg) == -1) {
    perror("semop: semctl(GETALL) failed");
  } else {
    fprintf(stderr, "Current semaphore values are:");
    for (i = 0; i < semid_ds.sem_nsems;
      fprintf(stderr, " %d", arg.array[i++]));
    fprintf(stderr, "\n");
  }
}
```

```
 /* Find out how many operations are going to be done in the
    next call and allocate enough space to do it. */
 fprintf(stderr, "How many semaphore operations do you want %s\n",
    "on the next call to semop()?");
 fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
  _exit(0);
 if (i > nsops) {
   if (nsops)
     free((char *)sops);
   nsops = i;
   if ((sops = (struct sembuf *)malloc((unsigned)
(nsops * sizeof(struct sembuf)))) == NULL) {
     fprintf(stderr, error_mesg2, nsops);
     _exit(2);
   }
 }
 *sopsp = sops;
 return (i);
```

```
}
```

```
/*****************************************
MODULO: semaph.c
SCOPO: Utilizzo di semafori
*****************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
union semun {
               int val;
               struct semid_ds *buf;
               ushort *array;
          };

main()
{ int i,j;
  int pid;
  int semid; /* semid of semaphore set */
  key_t key = 1234; /* key to pass to semget() */
```

```
int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
int nsems = 1; /* nsems to pass to semget() */
int nsops; /* number of operations to do */
struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf));
/* ptr to operations to perform */

/* set up semaphore */

 fprintf(stderr, "\nsemget: Setting up seamaphore:
  semget(%#lx, %\%#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
    perror("semget: semget failed");
    _exit(1);
 } else
    fprintf(stderr, "semget: semget succeeded: semid = %d\n", semid);

/* get child process */
if ((pid = fork()) < 0) {
    perror("fork");
    _exit(1);
```

```
}

if (pid == 0)  { /* child */
    i = 0;
    while (i  < 3) {/* allow for 3 semaphore sets */

      nsops = 2;

      /* wait for semaphore to reach zero */

      sops[0].sem_num = 0; /* We only use one track */
      sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
      sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */

      sops[1].sem_num = 0;
      sops[1].sem_op = 1; /* increment semaphore -- take control of track */
      sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

      /* Recap the call to be made. */
      fprintf(stderr,"\nsemop:Child Calling semop(%d,&sops,%d) \
```

```
                with:",semid,nsops);
        for (j = 0; j < nsops; j++) {
    fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
    fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
    fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}

        /* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
            perror("semop: semop failed");
}  else {
    fprintf(stderr, "\tsemop: semop returned %d\n", j);
            fprintf(stderr, "\n\nChild Process Taking Control of Track: \
                 %d/3 times\n", i+1);
    sleep(5); /* DO Nothing for 5 seconds */
        }
        nsops = 1;
        /* wait for semaphore to reach zero */
        sops[0].sem_num = 0;
        sops[0].sem_op = -1; /* Give UP COntrol of track */
```

```
    sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore, async */

    if ((j = semop(semid, sops, nsops)) == -1) {
       perror("semop: semop failed");
    }  else {
  fprintf(stderr, "Child Process Giving up Control of Track:
                %d/3 times\n", i+1);
    sleep(5);
        /* halt process to allow parent to catch semaphor change first */
    }
    ++i;
  }
}  else /* parent */  {
   i = 0;

   while (i  < 3) { /* allow for 3 semaphore sets */
     nsops = 2;
     /* wait for semaphore to reach zero */
     sops[0].sem_num = 0;
     sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
```

```
sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */

sops[1].sem_num = 0;
sops[1].sem_op = 1; /* increment semaphore -- take control of track */
sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

/* Recap the call to be made. */
fprintf(stderr,"\nsemop:Parent Calling semop(%d, &sops, %d) \
        with:", semid, nsops);
for (j = 0; j < nsops; j++) {
   fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j, sops[j].sem_num);
   fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
   fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
}
/* Make the semop() call and report the results. */
if ((j = semop(semid, sops, nsops)) == -1) {
  perror("semop: semop failed");
}  else {
  fprintf(stderr, "semop: semop returned %d\n", j);
  fprintf(stderr, "Parent Process Taking Control of Track: %d/3 times\n", i+
```

```
            sleep(5); /* Do nothing for 5 seconds */
            nsops = 1;

            /* wait for semaphore to reach zero */
            sops[0].sem_num = 0;
            sops[0].sem_op = -1; /* Give UP COntrol of track */
            sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT;
            /* take off semaphore, asynchronous  */
            if ((j = semop(semid, sops, nsops)) == -1) {
               perror("semop: semop failed");
          } else {
               fprintf(stderr, "Parent Process Giving up Control of Track:
                      %d/3 times\n", i+1);
               sleep(5);
               /* halt process to allow child to catch semaphor change first */
            }
            ++i;
         }
      }
   }
```

```
}
```

```
/*****************************************************
  NOME: sem1.c
  SCOPO: creazione di due semafori con potenziale deadlock
*****************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 75
#define S1 0
#define S2 0
int semid;
unsigned int count;
struct sembuf psembuf, vsembuf;

main (int argc, char **argv)
{
```

```
int i, sem1, sem2;
short initvec[2], outvec[2];

if (argc==1) {
  semid = semget(SEMKEY,2,0777|IPC_CREAT);
  initvec[0] = initvec[1] = 1;
  semctl(semid,SETALL,initvec);
  semctl(semid,GETALL,outvec);
  printf("Semaphore init values: %d %d\n",outvec[0],outvec[1]);
  pause();
} else if (!strcmp(argv[1],"0")) {
  sem1 = S1; sem2 = S2;
} else {
  sem1 = S2; sem2 = S1;
}


semid = semget(SEMKEY,2,0777);
psembuf.sem_op = -1; /* P */
psembuf.sem_flg = SEM_UNDO;
```

```
  vsembuf.sem_op = 1;  /* V */
  vsembuf.sem_op = SEM_UNDO;
  for (count=0;;count++) {
    psembuf.sem_num = sem1;
    semop(semid,&psembuf,1); /* P(s1) */
    psembuf.sem_num = sem2;
    semop(semid,&psembuf,1); /* P(s2) */
    printf("Proc %d - count %d\n", getpid(),count);
    vsembuf.sem_num = sem2;
    semop(semid,&vsembuf,1); /* V(s1) */
    vsembuf.sem_num = sem1;
    semop(semid,&vsembuf,1); /* V(s2) */
  }
}
```