ESD Electronic Systems Design
Synthesis Verification Testing Power Communication

Programmazione di sistema in Unix: Introduzione

N. Drago, G. Di Guglielmo, L. Di Guglielmo,
V. Guarnieri, M. Lora, G. Pravadelli, F. Stefanni

Introduzione

System call per il file system

Ulteriori system call per il file sistem – non oggetto d'esame

System call per la gestione dei processi

Introduzione

System call per il file system

Ulteriori system call per il file sistem – non oggetto d'esame

System call per la gestione dei 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 è 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*).

Alcune 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.
- È importante ottimizzare il numero di chiamate di sistema rispetto a quelle di libreria.
- Particolarmente evidente nel caso di system call per il file system.

Esempio:

```
1   /* PROG1 */
2   int main(void) {
3     int c;
4     while ((c = getchar()) != EOF) putchar(c);
5   }
6   /* PROG2 */
7   int main(void) {
8     char c;
9     while (read(0, &c, 1) > 0)
10    if(write(1, &c, 1)!=1){perror("write");exit(1)};
11  }
```

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

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

su `stderr` viene stampato:

```
1$ ./a.out
   str:  messaggio-di-errore \n
```

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

La libreria mylib

```
1  /*****************************************
2  MODULO: mylib.h
3  SCOPO: definizioni per la libreria mylib
4  *****************************************/
5
6  #ifndef MYLIB_H
7  #define MYLIB_H
8
9  void syserr (char *prog, char *msg);
10
11 #endif
```

La libreria mylib

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

#include "mylib.h"

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

Esempio di utilizzo di errno

```
1  #include <errno.h>
2  #include <unistd.h>
3  ...
4  /* Before calling the syscall, resetting errno */
5  errno = 0;
6  ssize_t bytes = write(1, "Hello!", 7);
7  if (bytes == -1)
8  {
9      if (errno == EBADF)
10     {
11         ...
12     }
13     ...
14 }
15 ...
```

Introduzione

- In UNIX esistono i seguenti tipi di file:
  1. File regolari
  2. Directory
  3. Link
  4. *pipe* o *fifo*
  5. *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` (`fopen()`, `fread()`, `fwrite()`, `fclose()`, `printf()`) sono tutte bufferizzate. Per efficienza, si può lavorare direttamente sui buffer.
- Le funzioni POSIX `open()`, `read()`, `write()`, `close()` sono non bufferizzate.
  - In questo caso i file non sono più descritti da uno *stream* ma da un *descrittore* (*file descriptor* – 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* anziché di file.

Apertura di un canale

```
1    #include <fcntl.h>
2
3    int open (char *name, int access, mode_t mode);
```

Valori del parametro `access` (vanno messi in OR):

- 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 (in OR):
`S_IRUSR S_IWUSR S_IXUSR S_IRGRP S_IWGRP S_IXGRP`
`S_IROTH S_IWOTH S_IXOTH S_IRWXU S_IRWXG S_IRWXO`
Corrispondenti ai modi di un file UNIX (u=RWX,g=RWX,o=RWX), e rimpiazzabili con codici numerici *ottali* (`0000 ... 0777`). Comunque, per portabilità e mantenibilità del codice, è sempre meglio usare le costanti fornite dallo standard.

Apertura di un canale

- Modi speciali di `open()`:
  - `O_EXCL`: apertura in modo esclusivo (nessun altro processo può aprire/creare)
  - `O_SYNC`: apertura in modo sincronizzato (file tipo lock, prima terminano eventuali operazioni di I/O in atto)
  - `O_TRUNC`: apertura di file esistente implica cancellazione contenuto
- Esempi di utilizzo:
  - `int fd = open("file.dat",O_RDONLY|O_EXCL,S_IRUSR);`
  - `int fd = open("file.dat",O_CREAT, S_IRUSR|S_IWUSR);`
  - `int fd = open("file.dat",O_CREAT, 0700);`

Apertura di un canale

```
1    #include <fcntl.h>
2
3    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.

Manipolazione diretta di un file

```
1     #include <unistd.h>
2
3     ssize_t read  (int fildes, void *buf, size_t n);
4     ssize_t write (int fildes, void *buf, size_t n);
5     int close (int fildes);
6
7     #include <sys/types.h>
8     #include <unistd.h>
9
10    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.
- `lseek()` riposiziona l'offset di un file aperto
  - Valori possibili di `whence`: `SEEK_SET` `SEEK_CUR` `SEEK_END`

Esempio

```
1  /*************************************
2  MODULO: lower.c
3  SCOPO: esempio di I/O non bufferizzato
4  *************************************/
5  #include <stdio.h>
6  #include <ctype.h>
7  #include "mylib.h"
8  #define BUFLEN 1024
9  #define STDIN 0
10 #define STDOUT 1
11
12 void lowerbuf (char *s, int l)
13 {
14   while (l-- > 0) {
15     if (isupper(*s)) *s = tolower(*s);
16     s++;
17   }
18 }
```

Esempio

```
1  int main (int argc, char *argv[])
2  {
3    char buffer[BUFLEN];
4    int x;
5
6    while ((x=read(STDIN,buffer,BUFLEN)) > 0)
7    {
8      lowerbuf (buffer, x);
9      x = write (STDOUT, buffer, x);
10     if (x == -1)
11       syserr (argv[0], "write() failure");
12   }
13   if (x != 0)
14     syserr (argv[0], "read() failure");
15   return 0;
16 }
```

Duplicazione di canali

```
1    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!

Introduzione

System call per il file system

Ulteriori system call per il file sistem – non oggetto d'esame

System call per la gestione dei processi

Creazione di una directory

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3
4    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

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	Ordinary file
	0000000	Ordinary file

- Per indicare il modo di esecuzione:

`S_ISUID`	0004000	Set user ID on execution
`S_ISGID`	0002000	Set group ID on execution
`S_ISVTX`	0001000	Set the sticky bit

Creazione di una directory

- 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

- 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:

```
1       #include <sys/stat.h>
2       #include <sys/types.h>
3       #include <fcntl.h>
4       #include <unistd.h>
5
6       int mkdir (const char *path, mode_t mode);
7       int rmdir (const char *path);
```

Accesso alle directory

- 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).

```
1    #include <sys/types.h>
2    #include <dirent.h>
3
4    DIR *opendir (char *dirname);
5    struct dirent *readdir (DIR *dirp);
6    void rewinddir (DIR *dirp);
7    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 alle directory

- Struttura interna di una directory:

```
1 struct dirent  {
2   __ino_t d_ino;                    /* inode # */
3   __off_t d_off;                    /* offset in the
4                                    directory structure */
5   unsigned short int d_reclen; /* how large this
6                                   structure really is */
7   unsigned char d_type;             /* file type */
8   char d_name[256];                 /* file name */
9 };
```

Esempio

```
1  /*************************************
2  MODULO: dir.c
3  SCOPO: ricerca in un directory e
4         rinomina di un file
5  *************************************/
6  #include <string.h>
7  #include <sys/types.h>
8  #include <sys/dir.h>
9
10 int dirsearch( char*, char*, char*);
11
12 int main (int argc, char *argv[])
13 {
14     return dirsearch (argv[1],argv[2],".");
15 }
```

Esempio

```
1  int dirsearch (char *file1, char* file2,  char *dir)
2  {
3    DIR *dp;
4    struct dirent *dentry;
5    int status = 1;
6
7    if ((dp=opendir (dir)) == NULL) return -1;
8    for (dentry=readdir(dp); dentry!=NULL;
9         dentry=readdir(dp))
10   if ((strcmp(dentry->d_name,file1)==0)) {
11     printf("Replacing entry %s with %s",
12            dentry->d_name,file2);
13     strcpy(dentry->d_name,file2);
14     return 0;
15   }
16   closedir (dp);
17   return status;
18 }
```

Accesso alle directory

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

- Cambia la directory corrente e si sposta in `dirname`.
- È necessario che la directory abbia il permesso di esecuzione.

Gestione dei Link

```
1    #include <unistd.h>
2
3    int link (char *orig_name, char *new_name);
4    int unlink (char *file_name);
```

- `link()` crea un hard link a `orig_name`. E' possibile fare riferimento al file con entrambi i nomi.
- `unlink()`
  - Cancella un file cancellando l'*i-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!

Esempio

```
1  /* Scopo: usare un file temporaneo senza che altri
2   * possano leggerlo.
3   * 1) aprire file in scrittura
4   * 2) fare unlik del file
5   * 3) usare il file, alla chiusura del processo
6   *      il file sara' rimosso
7   * NOTA: e' solo un esempio! Questo NON e' il modo
8   * corretto per creare file  temporanei. Per creare
9   * normalmente i file temporanei usare le funzioni
10  * tmpfile() o tmpfile64().
11  */
12  int   fd;
13  char  fname[32];
```

Esempio

```
1  ...
2  strcpy(fname,"myfile.xxx");
3  if ((fd = open(fname, O_WRONLY)) == -1)
4  {
5    perror(fname);
6    return 1;
7  } else if (unlink(fname) == -1) {
8    perror(fname);
9    return 2;
10 } else {
11   /* use temporary file */
12 }
13 ...
```

Privilegi e accessi

```
1    #include <unistd.h>
2    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

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    int chmod (char *file_name, int mode);
4    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	sticky bit
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

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    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

```
1    #include <sys/types.h>
2    #include <sys/stat.h>
3    #include <unistd.h>
4
5    int stat(char *file_name, struct stat *stat_buf);
6    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`:

```
1  dev_t           st_dev;      /* device */
2  ino_t           st_ino;      /* inode */
3  mode_t          st_mode;     /* protection */
4  nlink_t         st_nlink;    /* # of hard links */
5  uid_t           st_uid;      /* user ID of owner */
6  gid_t           st_gid;      /* group ID of owner */
7  dev_t           st_rdev;     /* device type
8                                * (if inode device) */
9  off_t           st_size;     /* total byte size */
10 unsigned long   st_blksize; /* blocksize for
11                               * filesystem I/O */
12 unsigned long   st_blocks;  /* number of blocks
13                               * allocated */
14 time_t          st_atime; /* time of last access */
15 time_t          st_mtime; /* time of last
16                             * modification */
17 time_t          st_ctime; /* time of last
18                            /* property change */
```

Esempio

```
1  #include <time.h>
2  ...
3  /* per stampare le informazioni con stat */
4  void display (char *fname, struct stat *sp)
5  {
6    printf ("FILE %s\n", fname);
7    printf ("Major number = %d\n", major(sp->st_dev));
8    printf ("Minor number = %d\n", minor(sp->st_dev));
9    printf ("File mode = %o\n", sp->mode);
10   printf ("i-node number = %d\n", sp->ino);
11   printf ("Links = %d\n", sp->nlink);
12   printf ("Owner ID = %d\n", sp->st_uid);
13   printf ("Group ID = %d\n", sp->st_gid);
14   printf ("Size = %d\n", sp->size);
15   printf ("Last access = %s\n", ctime(&sp->atime));
16 }
```

Stato di un file

- 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

```
1       #include <termio.h>
2       int ioctl(int fd, int request,
3                 ... /* argptr */ );
```

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

Le variabili di ambiente

```c
#include <stdlib.h>

char *getenv (char *env_var);
```

- Ritorna la definizione della variabile d'ambiente richiesta, oppure `NULL` se non è definita.
- Si può accedere anche alla seguente variabile globale:
  `extern char **environ;`
- È possibile esaminare in sequenza tutte le variabili d'ambiente usando il terzo argomento del `main()`. Questa signature non appartiene allo standard C, ma è diffusa nei sistemi POSIX. Pertanto è da evitare in programmi che vogliono essere portabili (usate uno degli altri due modi).

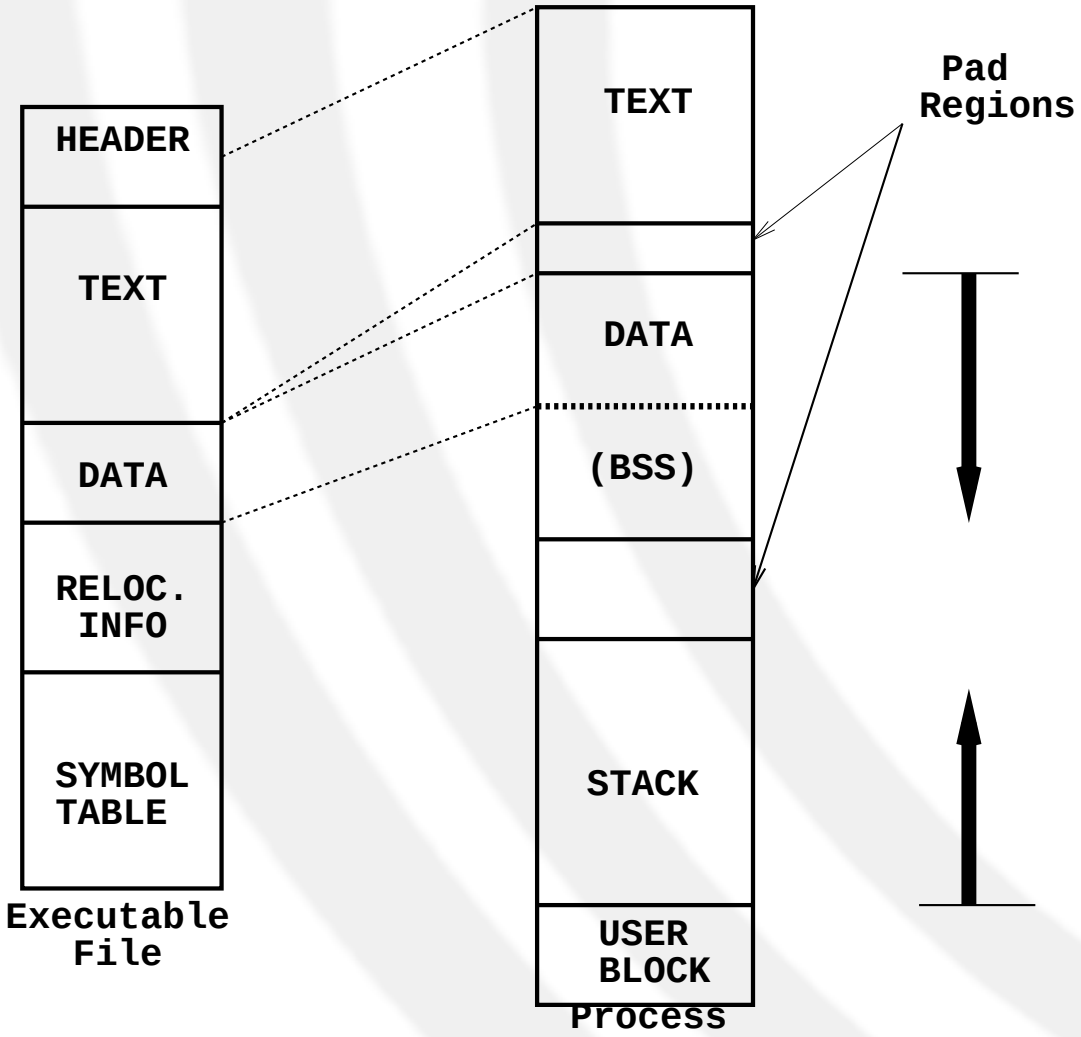
```c
int main (int argc, char *argv[], char *env[]);
```

Esempio

```
1  /*************************************
2  MODULO: env.c
3  SCOPO: elenco delle variabili d'ambiente
4  *************************************/
5  #include <stdio.h>
6
7  int main (int argc, char *argv[], char *env[])
8  {
9    puts ("Variabili d'ambiente:");
10   while (*env != NULL)
11     puts (*env++);
12   return 0;
13 }
```

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/linux/a.out.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 (statici, extern). 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: Visualizzabile con il comando `nm`. Può essere rimossa (`ld -s`) o con `strip` (programmi piú piccoli). Contiene informazioni quali la locazione, il tipo e lo scope di variabili, funzioni, tipi.

Gestione dei processi – Processo

- TEXT: copia di quello del programma eseguibile. 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 variabili d'ambiente
  - riallocato automaticamente dal sistema
  - cresce verso l'alto
- USER BLOCK: sottoinsieme delle informazioni mantenute dal sistema sul processo

Creazione di processi

```
1 #include <unistd.h>
2
3 pid_t fork (void)
```

- Crea un nuovo processo, figlio di quello corrente, che eredita dal padre:
  - I file aperti (non i lock)
  - Le variabili di ambiente
  - Directory di lavoro
- Solo la thread corrente viene replicata nel processo figlio.
- 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!

Creazione di processi - Esempio

```
1  /*************************************
2  MODULO: fork.c
3  SCOPO: esempio di creazione di un processo
4  *************************************/
5  #include <stdio.h>
6  #include <sys/types.h>
7  #include "mylib.h"
8  int main (int argc, char *argv[]){
9     pid_t status;
10    if ((status=fork()) == -1)
11    syserr (argv[0], "fork() fallita");
12    if (status == 0) {
13       sleep(10);
14       puts ("Io sono il figlio!");
15    } else {
16       sleep(2);
17       printf ("Io sono il padre e");
18       printf (" mio figlio ha PID=%d)\n", status);
19    }
20 }
```

Esecuzione di un programma

```
1  #include <unistd.h>
2  int execl (char *file,char *arg0,char *arg1, ...,
3      (char *) NULL)
4  int execlp(char *file,char *arg0,char *arg1, ...,
5      (char *) NULL)
6  int execle(char *file,char *arg0,char *arg1, ...,
7      (char *) NULL, char *envp[])
8  int execv (char *file,char *argv[])
9  int execvp(char *file,char *argv[])
10 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 (i.e., "interpreter script") avente come prima riga `#! interpreter`
- In altri termini, `exec` trasforma un eseguibile in processo.
- NOTA: `exec` non ritorna se il processo viene creato con successo!!

La Famiglia di `exec`

- crea un processo come se fosse stato lanciato direttamente da linea di comando (non vengono ereditati file descriptors, semafori, memorie condivise, etc.),
- `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`.
- `(char *) NULL` nella famiglia `execl` serve come terminatore nullo (come nel vettore `argv` del `main()`).
- Nella famiglia `execv`, gli array `argv` e `envp` devono essere terminati da NULL.

Esempio di chiamate ad exec

Esempio di chiamata ad `execl`:

```
1     execl("/bin/ls", "/bin/ls", "/tmp", (char*) NULL);
```

Esempio di chiamata a `execv`, equivalente al codice precedente:

```
1     char * argv[3];
2     argv[0] = "/bin/ls";
3     argv[1] = "/tmp";
4     argv[2] = NULL;
5     execv("/bin/ls", argv);
```

Esempio di loop su array NULL-terminated

Esempio di loop su argv tramite `while`:

```
1     int main( int argc, char * argv[] ) {
2       char ** s = argv;
3       while( *s != NULL )
4       {
5         char * curr = *s;
6         ...
7         ++s;
8       }
9     }
```

Esempio di loop su argv tramite `for`:

```
1     int main( int argc, char * argv[] ) {
2       for(char ** s = argv; *s != NULL; ++s)
3       {
4         char * curr = *s;
5         ...
6       }
7     }
```

Esecuzione di un programma - Esempio

```
1  /*************************************
2  MODULO: exec.c
3  SCOPO: esempio d'uso di exec()
4  *************************************/
5  #include <stdio.h>
6  #include <unistd.h>
7  #include "mylib.h"
8
9  int main (int argc, char *argv[])
10 {
11   puts ("Elenco dei file in /tmp");
12   execl ("/bin/ls", "/bin/ls", "/tmp", (char *) NULL);
13   syserr (argv[0], "execl() fallita");
14 }
```

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:

```
1  int pid = fork ();
2  if (pid == -1) {
3    perror("");
4  } else if (pid == 0) {
5    char *args [2];
6    args [0] = "ls"; args [1] = NULL;
7    execvp (args [0], args);
8    exit (1);    /* vedi dopo */
9  } else {
10   printf ("Sono il padre");
11   printf (" e mio figlio e' %d.\n", pid);
12 }
```

Sincronizzazione tra padre e figli

```
1 #include <sys/types.h>
2 #include <sys/wait.h>
3 #include <stdlib.h> // richiesto per exit ed _exit
4
5 void  exit(int status)
6 void  _exit(int status)
7 pid_t wait (int *status)
```

- `exit()` è un wrapper all'effettiva system call `_exit()`
- `exit()` chiude gli eventuali stream aperti.
- `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()` (a meno che il padre non abbia impostato di ignorare la terminazione dei figli).

Sincronizzazione tra padre e figli

- Lo stato può essere testato con le seguenti macro:

```
1 WIFEXITED(status)
2 WEXITSTATUS(status) // se WIFEXITED ritorna true
3 WIFSIGNALED(status)
4 WTERMSIG(status)    // se WIFSIGNALED ritorna true
5 WIFSTOPPED(status)
6 WSTOPSIG(status)    // se WIFSTOPPED ritorna true
```

- 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`

```
1 #include <sys/time.h>
2 #include <sys/resource.h>
3
4 pid_t waitpid (pid_t pid, int *status, int options)
5 pid_t wait3 (int *status, int options,
6               struct rusage *rusage)
```

- `waitpid` attende la terminazione di un particolare processo
  - `pid = -1`: tutti i figli
    - `wait(&status);` è equivalente a `waitpid(-1, &status, 0);`
  - `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` e' simile a `waitpid`, ma ritorna informazioni aggiuntive sull'uso delle risorse all'interno della struttura `rusage`. Vedere `man getrusage` per ulteriori informazioni.

Uso di wait() - Esempio - pt. 1

```
1  /*************************************
2  MODULO: wait.c
3  SCOPO: esempio d'uso di wait()
4  *************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <sys/wait.h>
9  #include <sys/types.h>
10 #include "mylib.h"
11
12 int main (int argc, char *argv[]){
13   pid_t child;
14   int status;
15
16   if ((child=fork()) == 0) {
17     sleep(5);
18     puts ("figlio 1 - termino con stato 3");
19     exit (3);
20   }
```

Uso di wait() - Esempio - pt. 2

```
1   if (child == -1)
2     syserr (argv[0], "fork() fallita");
3
4   if ((child=fork()) == 0) {
5     puts ("figlio 2 - sono in loop infinito,");
6     punts (" uccidimi con:");
7     printf (" kill -9 %d\n", getpid());
8
9     while (1) ;
10  }
11
12  if (child == -1)
13    syserr (argv[0], "fork() fallita");
```

Uso di `wait()` - Esempio - pt. 3

```
1   while ((child=wait(&status)) != -1) {
2     printf ("il figlio con PID %d e'", child);
3     if (WIFEXITED(status)) {
4       printf ("terminato (stato di uscita: %d)\n\n",
5         WEXITSTATUS(status));
6     } else if (WIFSIGNALED(status)) {
7       printf ("stato ucciso (segnale omicida: %d)\n\n",
8         WTERMSIG(status));
9     } else if (WIFSTOPPED(status)) {
10      puts ("stato bloccato");
11      printf ("(segnale bloccante: %d)\n\n",
12        WSTOPSIG(status));
13    } else if (WIFCONTINUED(status)) {
14      puts ("stato sbloccato");
15    } else
16      puts ("non c'e' piu' !?");
17  }
18  return 0;
19 }
```

Informazioni sui processi

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t uid = getpid()
5 pid_t gid = getppid()
```

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

Informazioni sui processi - Esempio

```
1  /*************************************************
2  MODULO: fork2.c
3  SCOPO: funzionamento di getpid() e getppid()
4  *************************************************/
5  #include <stdio.h>
6  #include <sys/types.h>
7  #include "mylib.h"
8  int main (int argc, char *argv[]) {
9    pid_t status;
10   if ((status=fork()) == -1) {
11     syserr (argv[0], "fork() fallita");
12   }
13   if (status == 0) {
14     puts ("Io sono il figlio:\n");
15     printf("PID = %d\tPPID = %d\n",getpid(),getppid());
16   }
17   else {
18     printf ("Io sono il padre:\n");
19     printf("PID = %d\tPPID = %d\n",getpid(),getppid());
20   }}
```

Informazioni sui processi – (cont.)

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 uid_t uid = getuid()
5 uid_t gid = getgid()
6 uid_t euid = geteuid()
7 uid_t egid = getegid()
```

- Ritornano la corrispondente informazione del processo corrente
- `geteuid` e `getegid` ritornano l'informazione sull'*effective* UID e GID.
- NOTA: la differenza tra *effective* e *real* sta nell'uso dei comandi `suid`/`sgid` che permettono di cambiare UID/GUID per permettere operazioni normalmente non concesse agli utenti. Il *real* UID/GID si riferisce sempre ai dati reali dell'utente che lancia il processo. L'*effective* UID/GID si riferisce ai dati ottenuti lanciando il comando `suid`/`sgid`.

Segnalazioni tra processi

- È possibile spedire asincronamente dei segnali ai processi:

```
1 #include <sys/types.h>
2 #include <signal.h>
3
4 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 con gruppo uguale a quello del processo chiamante
  - `(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.

```
1 #include <signal.h>
2 typedef void (*sighandler_t)(int);
3 sighandler_t signal(int signum,sighandler_t func);
```

- `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
- Il valore ritornato è l'handler registrato precedentemente.
- A seconda dell'implementazione o dei flag di compilazione, il comportamento cambia. Nelle versioni attuali di Linux:
  - Handler `SIG_DFL` o `SIG_IGN`: all'arrivo di un segnale, l'handler impostato viene mantenuto.
  - Flag di compilazione `-std=` o `-ansi`: all'arrivo di un segnale l'handler e' resettato a `SIG_DFL` (semantica Unix e System V).
  - Senza tali flag di compilazione: rimane impostato l'handler corrente (semantica BSD).

Segnalazioni tra processi – (cont.)

- Segnali disponibili (Linux): con il comando `kill` o su `man 7 signal`

POSIX.1-1990			
Signal	**Value**	**Def. Action**	**Description**
SIGHUP	1	Term	Hangup on contr. terminal or death of contr. process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Segnalazioni tra processi – (cont.)

POSIX.1-2001 and SUSv2			
Signal	**Value**	**Def. Action**	**Description**
SIGBUS	10,7,10	Core	Bus error (bad memory access)
SIGPOLL		Term	Pollable event (Sys V). Synonym for SIGIO
SIGPROF	27,27,29	Term	Profiling timer expired
SIGSYS	12,31,12	Core	Bad argument to routine (SVr4)
SIGTRAP	5	Core	Trace/breakpoint trap
SIGURG	16,23,21	Ign	Urgent condition on socket (4.2BSD)
SIGVTALRM	26,26,28	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	24,24,30	Core	CPU time limit exceeded (4.2BSD)
SIGXFSZ	25,25,31	Core	File size limit exceeded (4.2BSD)

Other signals			
Signal	**Value**	**Def. Action**	**Description**
SIGIOT	6	Core	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7	Term	
SIGSTKFLT	-,16,-	Term	Stack fault on coprocessor (unused)
SIGIO	23,29,22	Term	I/O now possible (4.2BSD)
SIGCLD	-,-,18	Ign	A synonym for SIGCHLD
SIGPWR	29,30,19	Term	Power failure (System V)
SIGINFO	29,-,-		A synonym for SIGPWR
SIGLOST	-,-,-	Term	File lock lost
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)
SIGUNUSED	-,31,-	Core	Synonymous with SIGSYS

Segnalazioni tra processi – (cont.)

- Dove multipli valori sono forniti, il primo è valido per alpha e sparc, quello centrale per ix86, ia64, ppc, s390, arm e sh, e l'ultimo per mips.
- Un trattino indica che il segnale non è presente sulle architetture specificate.
- I segnali SIGKILL e SIGSTOP non possono essere intercettati.
- I processi figlio ereditano le impostazioni dal processo padre, ma `exec` resetta ai valori di default.
- NOTA: in processi multithread, la ricezione di un segnale risulta in un comportamento *non definito*!

Segnalazioni tra processi - Esempio 1 - pt. 1

```
1  #include <stdio.h>    /* standard I/O functions          */
2  #include <unistd.h>   /* standard unix functions,        *
3                         * like getpid()                   */
4  #include <signal.h>   /* signal name macros, and the     *
5                         * signal() prototype              */
6
7  /* first, here is the signal handler */
8  void catch_int(int sig_num)
9  {
10     /* re-set the signal handler again to catch_int,
11      * for next time */
12     signal(SIGINT, catch_int);
13     printf("Don't do that\n");
14     fflush(stdout);
15 }
```

Segnalazioni tra processi - Esempio 1 - pt. 2

```
1  int main(int argc, char* argv[])
2  {
3      /* set the INT (Ctrl-C) signal handler
4       * to 'catch_int' */
5      signal(SIGINT, catch_int);
6
7      /* now, lets get into an infinite loop of doing
8       * nothing. */
9      for ( ;; )
10         pause();
11 }
```

Segnalazioni tra processi - Esempio 2 - pt. 1

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

long maxprim = 0;
long np=0;

void usr12_handler (int s) {
  printf ("\nRicevuto segnale n.%d\n",s);
  printf ("Il piu' grande primo trovato e'");
  printf ("%ld\n",maxprim);
  printf ("Totale dei  numeri primi=%d\n",np);
}
```

Segnalazioni tra processi - Esempio 2 - pt. 2

```
1  void good_bye (int s) {
2    printf ("\nIl piu' grande primo trovato e'");
3    printf ("%ld\n",maxprim);
4    printf ("Totale dei  numeri primi=%d\n",np);
5    printf ("Ciao!\n");
6    exit (1);
7  }
8
9  int is_prime (long x) {
10   long fatt;
11   long maxfatt = (long)ceil(sqrt((double)x));
12   if (x < 4) return 1;
13   if (x % 2 == 0) return 0;
14
15   for (fatt=3; fatt<=maxfatt; fatt+=2)
16     return (x % fatt == 0 ? 0: 1);
17 }
```

Segnalazioni tra processi - Esempio 2 - pt. 3

```
1  int main (int argc, char *argv[]) {
2    long n;
3
4    signal (SIGUSR1, usr12_handler);
5    /* signal (SIGUSR2, usr12_handler); */
6    signal (SIGHUP, good_bye);
7
8    printf("Usa kill -SIGUSR1 %d per vedere il numero
9            primo corrente\n", getpid());
10   printf("Usa kill -SIGHUP %d per uscire", getpid());
11   fflush(stdout);
12
13   for (n=0; n<LONG_MAX; n++)
14   if (is_prime(n)) {
15     maxprim = n;
16     np++;
17   }
18 }
```

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 lo sblocco della `wait()`)
- Può essere intercettato per modificare l'azione corrispondente

Timeout e Sospensione

```
1 #include <unistd.h>
2 unsigned int alarm (unsigned seconds)
```

- `alarm` invia un segnale (SIGALRM) al processo chiamante dopo `seconds` secondi. Se `seconds` vale 0, l'allarme è annullato.
- 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 - Esempio - pt. 1

```
1  #include <stdio.h>    /* standard I/O functions          */
2  #include <unistd.h>   /* standard unix functions,        *
3                         * like getpid()                   */
4  #include <signal.h>   /* signal name macros, and the     *
5                         * signal() prototype              */
6
7  /* buffer to read user name from the user */
8  char user[40];
9
10 /* define an alarm signal handler. */
11 void catch_alarm(int sig_num)
12 {
13     printf("Operation timed out. Exiting...\n\n");
14     exit(0);
15 }
```

Timeout e sospensione - Esempio - pt. 2

```c
int main(int argc, char* argv[])
{
    /* set a signal handler for ALRM signals */
    signal(SIGALRM, catch_alarm);

    /* prompt the user for input */
    printf("Username: ");
    fflush(stdout);
    /* start a 10 seconds alarm */
    alarm(10);
    /* wait for user input */
    scanf("%s",user);
    /* remove the timer, now that we've got
     * the user's input */
    alarm(0);

    printf("User name: '%s'\n", user);
    return 0;
}
```

Timeout e Sospensione - (cont.)

```
1 #include <unistd.h>
2 int pause ()
```

- Sospende un processo fino alla ricezione di un qualunque segnale.
- Ritorna sempre -1
- N.B.: se si usa la `alarm` per uscire da una `pause` bisogna inserire l'istruzione `alarm(0)` dopo la `pause` per disabilitare l'allarme. Questo serve per evitare che l'allarme scatti dopo anche se `pause` e' già uscita a causa di un'altro segnale.

Introduzione

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

UNIX e IPC

- `ipcs`: riporta lo stato di tutte le risorse, o selettivamente, con le seguenti opzioni:
  - `-s` informazioni sui semafori;
  - `-m` informazioni sulla memoria condivisa;
  - `-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`

UNIX e IPC - (cont.)

- 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*:

```
1 #include <unistd.h>
2
3 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.

Pipe - Esempio - part. 1

```
1  /*************************************
2  MODULO: pipe.c
3  SCOPO: esempio di IPC mediante pipe
4  *************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8  #include <sys/wait.h>
9
10 int main (int argc, char *argv[]) {
11   int status, p[2];
12   char buf[64];
13
14   pipe (p);
15   if ((status=fork()) == -1)   /* errore */
16     syserr (argv[0], "fork() fallita");
```

Pipe - Esempio - part. 2

```
1    else if (status == 0) { /* figlio */
2      close (p[1]);
3      if (read(p[0],buf,BUFSIZ) == -1)
4        syserr (argv[0], "read() fallita");
5      printf ("Figlio - ricevuto: %s\n", buf);
6      exit(0);
7    } else  {  /* padre */
8      close(p[0]);
9      printf("Padre - invio nella pipe:");
10     printf("In bocca al lupo\n");
11     write(p[1], "In bocca al lupo", 17);
12     wait(&status);
13     exit(0);
14   }
15 }
```

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).
- La dimensione è definita in `PIPE_BUF`.
- 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 - Esempio - part. 1

```
1  /*************************************************
2  MODULO: pipe2.c
3  SCOPO: Realizzare il comando "ps | sort"
4  *************************************************/
5  #include <sys/types.h>
6  #include <stdlib.h>
7  #include <unistd.h>
8
9  int main ()
10 {
11         pid_t pid;
12         int pipefd[2];
```

Pipe e comandi - Esempio - part. 2

```
1           pipe (pipefd);
2           if ((pid = fork()) == 0) { /* figlio */
3                 close(1);       /* close stdout */
4                 dup (pipefd[1]);
5                 close (pipefd[0]);
6                 execlp ("ps", "ps", NULL);
7           }
8           else if (pid > 0) { /* padre */
9                 close(0); /* close stdin  */
10                dup (pipefd[0]);
11                close (pipefd[1]);
12                execlp ("sort", "sort", NULL);
13          }
14          return 0;
15    }
```

Pipe e I/O non bloccante

- E' possibile forzare il comportamento di `write` e `read` rimuovendo la limitazione del bloccaggio.
- Realizzato tipicamente con `fcntl` per impostare la flag `O_NONBLOCK` sul corrispondente file descriptor (0 o 1)

```
1 int fd[2];
2 fcntl(fd[0], F_SETFL, O_NONBLOCK);
```

- Utile per implementare meccanismi di polling su pipe.
  - Se la flag `O_NONBLOCK` è impostata, una `write` su una pipe piena ritorna subito 0, e una `read` su una pipe vuota ritorna subito 0.

Pipe - (cont.)

- 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 anziché attraverso il file descriptor.
- Esistono fisicamente su disco e devono essere rimossi esplicitamente con `unlink`

Named Pipe (FIFO)

```
1 int mknod(const char *pathname,mode_t mode,dev_t dev);
2 // Esempio:
3 char* name = "my_fifo";
4 int result = mknod (name, S_IFIFO |
5                                 S_IRUSR | S_IWUSR, 0);
```

- Si creano con `mknod()`. L'argomento `dev` è ignorato.
- Valore di ritorno: 0 in caso di successo, -1 in caso di errore.
- Apertura, lettura/scrittura, chiusura avvengono come per un normale file. Quindi sono ereditate dai processi figlio.
- 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` e flag `O_NONBLOCK`) un FIFO in modo non bloccante. In tal modo sia `read` che `write` saranno non bloccanti.

Named Pipe (FIFO) - (cont.)

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 int mkfifo(const char *pathname, mode_t mode);
```

- A livello di libreria (non system call) esiste anche la funzione `mkfifo()`.
- Il valore di ritorno e i parametri sono gli stessi di `mknod()`.

Named Pipe (FIFO) - Example - part. 1

```
1  /*************************************
2  MODULO: fifo.c
3  SCOPO: esempio di IPC mediante named pipe
4
5  USO: Lanciare due copie del programma su due
6  shell separate, una con flag -0 e
7  l'altra con flag -1.
8  Lanciare prima quella con flag -1 che crea
9  la fifo.
10 La copia del programma con flag -0 leggera'
11 quanto scritto dalla copia con flag -1.
12 *************************************/
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <fcntl.h>
16 #include <sys/types.h>
17 #include <sys/stat.h>
18 #include <unistd.h>
```

Named Pipe (FIFO) - Example - part. 2

```
1  int main (int argc, char *argv[]) {
2    int i,fd;
3    char buf[64];
4
5    if (argc != 2) {
6      printf("Usage: fifo.x -[0|1]\n\t -0 to");
7      printf(" read\n\t -1 to write\n");
8      exit(1);
9    } else if (strncmp(argv[1], "-1", 2)==0){
10     if (mknod("fifo",S_IFIFO|0777,0)== -1) {
11       perror("mknod");
12       exit(1);
13     }
14     if ((fd = open("fifo",O_WRONLY))==-1){
15       perror("FIFO: -1");
16       unlink("fifo");
17       exit(1);
18     }
```

Named Pipe (FIFO) - Example - part. 3

```
1   } else if (strncmp(argv[1], "-O", 2)==0){
2     if ((fd = open("fifo",O_RDONLY))==-1){
3       perror("FIFO: -1");
4       unlink("fifo");
5       exit(1);
6     }
7   } else {
8     printf("Wrong parameter: %s\n", argv[1]);
9     unlink("fifo");
10    exit(1);
11  }
12  for (i=0;i<20;i++) {
13    if (strncmp(argv[1], "-1", 2)==0){
14      write(fd,"HELLO",6);
15      printf("Written HELLO %d \n", i);
16    } else {
17      read(fd,buf,6);
18      printf("Read %s %d\n",buf,i);}}}
```

Meccanismi di IPC Avanzati

- IPC SystemV:
  - Code di messaggi
  - Memoria condivisa
  - Semafori
- Caratteristiche comuni:
  - Una primitiva "`get`" per:
    - creare una nuova entry,
    - recuperare una entry esistente
  - Una primitiva "`ctl`" (control) per:
    - verificare lo stato di una entry,
    - cambiare lo stato di una entry,
    - rimuovere una entry.
  - Sono visibili e manipolabili tramite i comandi bash `ipcs` (anche per controllare le chiavi) e `ipcrm` (in caso di crash del programma).

Meccanismi di IPC Avanzati - (cont.)

- La primitiva "get" richiede la specifica di due informazioni:
  - Una *chiave*, usata per la creazione/recupero dell'oggetto di sistema
    - Valore intero arbitrario;
  - Dei *flag* di utilizzo:
    - Permessi relativi all'accesso (tipo `rwxrwxrwx`)
    - `IPC_CREAT`: si crea una nuova entry se la chiave non esiste
    - `IPC_CREAT` + `IPC_EXCL`: si crea una nuova entry ad uso esclusivo da parte del processo

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, che varia a seconda dei 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`:

```
struct ipc_perm {
    key_t key; /* Key. */
    uid_t uid; /* Owner 's user ID. */
    gid_t gid; /* Owner 's group ID. */
    uid_t cuid ; /* Creator 's user ID. */
    gid_t cgid ; /* Creator 's group ID. */
    unsigned short int mode ; /* R/W perm. */
}
```

Meccanismi di IPC Avanzati - (cont.)

- La primitiva "`ctl`" richiede la specifica di informazioni diverse in base all'oggetto di sistema
- In tutti i casi, la primitiva "`ctl`" richiede:
  - Un *descrittore*, usato per accedere all'oggetto di sistema
    - Valore intero ritornato dalla primitiva "`get`";
  - Dei *comandi* di utilizzo:
    - Cancellare
    - Modificare
    - Leggere informazioni relative agli oggetti

ftok()

```
1 #include <sys/types.h>
2 #include <sys/ipc.h>
3
4 key_t ftok(const char *pathname, int proj_id);
```

- `ftok()` è usata per ottenere chiavi probabilmente non in uso nel sistema.
- Utile per evitare possibili conflitti con altri processi. È migliore usare `ftok()` rispetto a delle costanti esplicite.
- I parametri sono un path ad un file esistente ed accessibile, e una costante espressa su un *byte* (sugli 8 bit meno significativi).
- Ritorna -1 in caso di errore, o una chiave univoca a parità di path e identificativo.

Code di Messaggi

- Un messaggio è una unità di informazione di dimensione variabile, senza un formato predefinito.
- Una *coda* è un oggetto di sistema, identificato da una chiave *key*, contenente uno o più messaggi.

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

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

- Serve a ottenere l'identificatore di una coda di messaggi se trova una corrispondenza, altrimenti restituisce un errore;
- Serve a creare una coda di messaggi data la chiave `key` nel caso in cui:
  - `key` = `IPC_PRIVATE`, oppure
  - `key` $\neq$ `IPC_PRIVATE` e `flag & IPC_CREAT` è vero.

Code di Messaggi: Gestione

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

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

- La funzione permette di accedere ai valori della struttura *msqid_ds*, mantenuta all'indirizzo *buf*, per la coda specificata dalla chiave *id*. `id` è il descrittore ritornato da `msgget`
- Il comportamento della funzione dipende dal valore dell'argomento *cmd*, che specifica il tipo di azione da eseguire:
  - `IPC_RMID`: cancella la coda (`buffer` non è 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.)

- `buf` 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 #bytes on queue */
  msgqnum_t msg_qnum;       /* #messages currently on queue */
  msglen_t msg_qbytes;      /* max #bytes allowed on queue */
  __pid_t msg_lspid;        /* pid of last msgsnd() */
  __pid_t msg_lrpid;        /* pid of last msgrcv() */
};
```

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)
```

- La funzione invia un messaggio sulla coda `id` (`id` è il descrittore ritornato da `msgget`)
- Il messaggio ha lunghezza specificata da `size`, ed è passato attraverso l'argomento `msg`
- La funzione restituisce `0` in caso di successo, oppure `-1` in caso di errore
- Struttura `msgbuf` dei messaggi:

```
struct msgbuf {
    long    mtype;      /* message type > 0 */
    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.)

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

int msgrcv(int id, struct msgbuf *msg, size_t size, long type, int flag);
```

- Legge un messaggio dalla coda `id`, lo scrive sulla struttura puntata da `msg` e ritorna il numero di byte letti. Una volta estratto, il messaggio sarà rimosso dalla coda
- L'argomento `size` indica la lunghezza massima del testo del messaggio
- Se il testo del messaggio ha lunghezza superiore a `size` il messaggio non viene estratto e la funzione ritorna con un errore.

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`

Code di Messaggi: esempi di gestione del messaggio

```c
typedef struct {     /* Example 1: fixed length string */
    long mtype;
    char mtext[256];
} Msg1; Msg1 msg1;
const char * text = "Hello!";
strcpy(msg1.mtext, text); msg1.mtype = 1;
msgsnd(mq_id, &msg1, sizeof(Msg1) - sizeof(long), 0);
```

```c
typedef struct Msg2{ /* Example 2: multiple types */
    long mtype;
    int id;
    char s[22];
    int i;
} Msg2; Msg2 msg2; msg2.mtype = 2;
msgsnd(mq_id, &msg2, sizeof(Msg2) - sizeof(long), 0);
```

```c
typedef struct Msg3{ /* Example 3: variable length */
    long mtype;
    char mtext[1];
} Msg3; Msg3 * msg3;
const char * text = "Hello!";
msg3 = (Msg3) malloc(sizeof(Msg) + strlen(text)*sizeof(char));
strcpy(msg3->mtext, text); msg3->mtype = 3;
msgsnd(mq_id,msg3,
  sizeof(Msg)+strlen(text)*sizeof(char)-sizeof(long),0);
```

Code di Messaggi: Esempio (Server - part 1)

```
1  /*************************************
2    PROCESSO SERVER
3  *************************************/
4  #include <sys/types.h>
5  #include <unistd.h>
6  #include <sys/wait.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define MSGKEY  75
13 #define MSGTYPE  1
14
15 int main (int argc,char *argv[]) {
16   key_t msgkey;
17   int msgid, pid;
```

Code di Messaggi: Esempio (Server - part 2)

```
1   struct msg {
2     long mtype;
3     char mtext[256];
4   } Message;
5
6   if ((msgid = msgget(MSGKEY,(0666|IPC_CREAT|IPC_EXCL))) == -1) {
7     perror(argv[0]);
8   }
9   const unsigned msg_size = sizeof(msg) - sizeof(long);
10  /* leggo dalla coda, aspettando il primo messaggio */
11  msgrcv(msgid,&Message,msg_size,MSGTYPE,0);
12  printf("Received from client: %s\n",Message.mtext);
13
14  /* scrivo in un messagio il pid e lo invio*/
15  pid = getpid();
16  sprintf(Message.mtext,"%d",pid);
17  Message.mtype = MSGTYPE;
18  msgsnd(msgid,&Message,msg_size,0);   /* WAIT */
19 }
```

Code di Messaggi: Esempio (Client - part 1)

```
1  /*************************************
2    PROCESSO CLIENT
3  *************************************/
4  #include <sys/types.h>
5  #include <unistd.h>
6  #include <sys/wait.h>
7  #include <sys/ipc.h>
8  #include <sys/msg.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11
12 #define MSGKEY  75
13 #define MSGTYPE  1
14
15 int main (int argc,char *argv[]) {
16   key_t msgkey;
17   int msgid, pid;
```

Code di Messaggi: Esempio (Client - part 2)

```
1   typedef struct msg {
2     long mtype;
3     char mtext[256];
4   } msg;
5   msg Message;
6
7   if ((msgid = msgget(MSGKEY,0666)) == -1) {
8     perror(argv[0]);
9   }
10  /* invio il PID in un messagio */
11  pid = getpid();
12  sprintf(Message.mtext,"%d",pid);
13  const unsigned msg_size = sizeof(msg)-sizeof(long);
14  Message.mtype = MSGTYPE;
15  msgsnd(msgid,&Message,msg_size,0);   /* WAIT */
16
17  /* Ricevo il messaggio del server -- wait */
18  msgrcv(msgid,&Message,msg_size,MSGTYPE,0);
19  printf("Received message from server: %s\n",Message.mtext);
20 }
```

Code di Messaggi: Esempio (Controller - part 1)

```
1  /*****************************************
2  MODULO: msgctl.c
3  SCOPO: Illustrare il funz. di msgctl()
4  *****************************************/
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>
10 #include <time.h>
11
12 void do_msgctl();
13 char warning_message[] = "If you remove read permission for"
14                          "yourself, this program will fail frequently!";
15
16 int main(int argc, char* argv[]) {
17   struct msqid_ds buf; /*buffer per msgctl()*/
18   int cmd;    /* comando per msgctl() */
19   int msqid; /* ID della coda da passare a msgctl() */
```

Code di Messaggi: Esempio (Controller - part 2)

```
1   if (argc!=2){
2     printf("Usage: msgctl.x <msgid>\n");
3     exit(1);
4   }
5
6   msqid = atoi(argv[1]);
7
8   fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
9   fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
10  fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
11  fprintf(stderr, "\nScegliere l'opzione desiderata: ");
12
13  scanf("%i", &cmd);
```

Code di Messaggi: Esempio (Controller - part 3)

```
1   switch (cmd) {
2   case IPC_SET:
3     fprintf(stderr, "Prima della IPC_SET,
4                      controlla i valori correnti:");
5     /* notare: non e' stato inserito il break, quindi di seguito
6     vengono eseguite le istruzioni del case IPC_STAT */
7
8   case IPC_STAT:
9     do_msgctl(msqid, IPC_STAT, &buf);
10    fprintf(stderr, "msg_perm.uid = %d\n", buf.msg_perm.uid);
11    fprintf(stderr,"msg_perm.gid = %d\n", buf.msg_perm.gid);
12    fprintf(stderr, "msg_perm.cuid = %d\n", buf.msg_perm.cuid);
13    fprintf(stderr, "msg_perm.cgid = %d\n", buf.msg_perm.cgid);
14    fprintf(stderr, "msg_perm.mode = %#o, ", buf.msg_perm.mode);
15    fprintf(stderr, "access permissions = %#o\n",
16                    buf.msg_perm.mode & 0777);
17    fprintf(stderr, "msg_cbytes = %d\n", buf.msg_cbytes);
18    fprintf(stderr, "msg_qbytes = %d\n",buf.msg_qbytes);
19    fprintf(stderr, "msg_qnum = %d\n", buf.msg_qnum);
20    fprintf(stderr, "msg_lspid = %d\n", buf.msg_lspid);
21    fprintf(stderr, "msg_lrpid = %d\n", buf.msg_lrpid);
```

Code di Messaggi: Esempio (Controller - part 4)

```
1     if (buf.msg_stime) {
2       fprintf(stderr, "msg_stime = %s\n", ctime(&buf.msg_stime));
3     }
4     if (buf.msg_rtime) {
5       fprintf(stderr, "msg_rtime = %s\n", ctime(&buf.msg_rtime));
6     }
7       fprintf(stderr, "msg_ctime = %s", ctime(&buf.msg_ctime));
8
9     /* se il comando originario era IPC_STAT allora esco
10       dal case, altrimenti proseguo con le operazioni
11       specifiche per la modifica dei parametri della coda.
12    */
13    if (cmd == IPC_STAT)
14      break;
```

Code di Messaggi: Esempio (Controller - part 5)

```
1       /* Modifichiamo alcuni parametri della coda */
2       fprintf(stderr, "Enter msg_perm.uid: ");
3       scanf ("%hi", &buf.msg_perm.uid);
4       fprintf(stderr, "Enter msg_perm.gid: ");
5       scanf("%hi", &buf.msg_perm.gid);
6       fprintf(stderr, "%s\n", warning_message);
7       fprintf(stderr, "Enter msg_perm.mode: ");
8       scanf("%hi", &buf.msg_perm.mode);
9       fprintf(stderr, "Enter msg_qbytes: ");
10      scanf("%hi", &buf.msg_qbytes);
11      do_msgctl(msqid, IPC_SET, &buf);
12      break;
13
14    case IPC_RMID:
15    default:
16      /* Rimuove la coda di messaggi. */
17      do_msgctl(msqid, cmd, (struct msqid_ds *)NULL);
18      break;
19    }
20    exit(0);
21  }
```

Code di Messaggi: Esempio (Controller - part 6)

```
1  void do_msgctl(int msqid, int cmd, struct msqid_ds* buf) {
2    int rtrn; /* per memorizzare il valore di ritorno della msgctl() */
3
4    fprintf(stderr, "\nmsgctl: Calling msgctl(%d, %d, %s)\n",
5            msqid, cmd, buf ? "&buf" : "(struct msqid_ds *)NULL");
6    rtrn = msgctl(msqid, cmd, buf);
7
8    if (rtrn == -1) {
9      perror("msgctl: msgctl failed");
10     exit(1);
11   } else {
12     fprintf(stderr, "msgctl: msgctl returned %d\n", rtrn);
13   }
14 }
```

Analisi Esempi

Per provare gli esempi precedenti:

- lanciare il programma server
- lanciare il programma client su una shell separata
  - client e server si scambieranno un messaggio e termineranno lasciando la coda di messaggi attiva.
- eseguire il comando `ipcs -q` per verificare che effettivamente esista una coda attiva.
- lanciare il programma `msgctl.c` passandogli come parametro il valore *msqid* visualizzato dal comando `ipcs -q`
- provare le varie opzioni del programma `msgctl.c`, in particolare usare `IPC_SET` per variare le caratteristiche della coda e `IPC_RMID` per rimuovere la coda

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>

int shmget(key_t key, size_t 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>

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

- `shmaddr` indica l'indirizzo virtuale dove il processo vuole `attaccare` il segmento di memoria condivisa. Tipicamente è `NULL`.
- Il valore di ritorno rappresenta l'indirizzo di memoria condivisa effettivamente risultante.
- La memoria è ereditata dai processi creati con `fork()`, ma non con `exec()`.
- In caso di errore ritorna `(void *) -1`.

Memoria Condivisa – (cont.)

- In base ai valori di `flag` e di `shmaddr` si determina il punto di attacco del segmento:
  - shmaddr == NULL: memoria attaccata in un indirizzo a scelta del sistema operativo.
  - shmaddr != NULL && (flag & SHM_RND): attaccata al multiplo di `SHMLBA` più vicino a shmaddr, ma non maggiore di shmaddr.
  - shmaddr != NULL && !(flag & SHM_RND): shmaddr deve essere allineato ad una pagina di memoria.
- Il segmento è attaccato in lettura se `flag & SHM_RDONLY` è vero, altrimenti è contemporaneamente in lettura e scrittura.
- Un segmento attaccato in precedenza può essere "staccato" (*detached*) con `shmdt`

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

- `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

```c
#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 all'area di memoria condivisa nella struttura puntata da `buffer` (contiene info su UID, GID, permessi, stato della memoria).
  - `IPC_SET`: modifica un sottoinsieme dei campi contenuti nella struct (UID, GID, permessi).
  - `SHM_LOCK`: impedisce che il segmento venga swappato o paginato.
  - `SHM_UNLOCK`: ripristina il normale utilizzo della memoria condivisa

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 */
```

Memoria Condivisa: esempi accesso

```
1  int shmid1 = shmget(key1,sizeof(int),0666|IPC_CREAT);
2  int * i1 = (int*) shmat(shmid1, NULL, 0);
3
4  int shmid2 = shmget(key2,sizeof(int) * 10,0666|IPC_CREAT);
5  int * i2 = (int*) shmat(shmid2, NULL, 0);
6
7  typedef struct Data { /* Layout: memoria contigua! */
8    int i1;
9    char buf[20];
10   int i2;
11 } Data;
12 int shmid3 = shmget(key3,sizeof(Data),0666|IPC_CREAT);
13 Data * d = (Data*) shmat(shmid3, NULL, 0);
14
15 typedef struct Msg { /* As mqueue */
16   long type;
17   char buf[1];
18 } Msg;
19 int shmid4 = shmget(key4,sizeof(Msg)+sizeof(char)*256,0666|IPC_CREAT);
20 Msg * d = (Msg*) shmat(shmid4, NULL, 0);
```

Memoria Condivisa: xmalloc

- Esempio di come creare delle funzioni simili a `malloc()` e `free()`, per la memoria condivisa.

```
1  typedef struct XMem {
2    key_t key;
3    int shmid;
4    char buf[1];
5  } XMem;
6
7  void * xmalloc( key_t key, const size_t size) {
8    const int shmid = shmget(key,size+sizeof(XMem)-sizeof(char),
9      0666|IPC_CREAT);
10   if (shmid == -1) return NULL;
11   XMem * ret = (XMem*) shmat(shmid, NULL, 0);
12   if (ret == (void*)-1) return NULL;
13   ret->key = key;
14   ret->shmid = shmid;
15   return ret->buf;
16 }
```

Memoria Condivisa: xmalloc – (cont.)

```
1  void xfree(void * ptr) {
2    XMem tmp;
3    XMem * mem = (XMem *)(((char*)ptr)
4      - (((char*)&tmp.buf) - ((char*)&tmp.key)));
5    const int shmid = mem->shmid;
6    shmdt(mem);
7    shmctl(shmid, IPC_RMID, NULL);
8  }
9
10 int main(int argc, char * argv[]) {
11   int * xi = (int*) xmalloc(ftok(argv[0], 'a'), sizeof(int) * 8);
12   ...
13   xfree(xi);
14 }
```

Memoria Condivisa: Esempio (Controller - part 1)

```
1  /*****************************************************
2  MODULO: shmctl.c
3  SCOPO: Illustrare il funz. di shmctl()
4  USO: Lanciare il programma e fornire l'ID
5       di un segmento di memoria condivisa
6       precedentemente creato.
7       Usare il comando della shell ipcs per vedere
8       i segmenti di memoria condivisa attivi
9  *****************************************************/
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <sys/types.h>
13 #include <sys/ipc.h>
14 #include <sys/shm.h>
15 #include <time.h>
16
17 void do_shmctl();
```

Memoria Condivisa: Esempio (Controller - part 2)

```
1  int main(int argc, char *argv[]) {
2    int cmd; /* comando per shmctl() */
3    int shmid; /* ID dell'area di memoria condivisa*/
4    struct shmid_ds shmid_ds; /* struttura per il controllo
5                                 dell'area di memoria condivisa*/
6
7    fprintf(stderr,"Inserire l'ID del segmento di memoria condiviso:");
8    scanf("%i", &shmid);
9
10   fprintf(stderr, "Comandi validi:\n");
11   fprintf(stderr, "\tIPC_RMID =\t%d\n", IPC_RMID);
12   fprintf(stderr, "\tIPC_SET =\t%d\n", IPC_SET);
13   fprintf(stderr, "\tIPC_STAT =\t%d\n", IPC_STAT);
14   fprintf(stderr, "\tSHM_LOCK =\t%d\n", SHM_LOCK);
15   fprintf(stderr, "\tSHM_UNLOCK =\t%d\n", SHM_UNLOCK);
16   fprintf(stderr, "Scegliere il comando desiderato: ");
17   scanf("%i", &cmd);
```

Memoria Condivisa: Esempio (Controller - part 3)

```
1   switch (cmd) {
2     case IPC_STAT:
3       /* Le informazioni sullo stato della memoria condivisa
4       vengono recuperate con la chiamata alla funzione
5       do_shmctl(shmid, cmd, &shmid_ds) eseguita al termine
6       del case. Le informazioni saranno inserite nella
7       struttura shmid_ds*/
8       break;
9     case IPC_SET:
10      /*visualizzazione dello stato attuale della memoria */
11      do_shmctl(shmid, IPC_STAT, &shmid_ds);
12      /* Lettura da tastiera dei valori */
13    /* di UID, GID, e permessi da settare */
14      fprintf(stderr, "\nInserire shm_perm.uid: ");
15      scanf("%hi", &shmid_ds.shm_perm.uid);
16      fprintf(stderr, "Inserire shm_perm.gid: ");
17      scanf("%hi", &shmid_ds.shm_perm.gid);
18      fprintf(stderr, "N.B.: Mantieni il permesso
19              di lettura per te stesso!\n");
20      fprintf(stderr, "Inserire shm_perm.mode: ");
21      scanf("%hi", &shmid_ds.shm_perm.mode);
22      break;
```

Memoria Condivisa: Esempio (Controller - part 4)

```
1       case IPC_RMID: /* Rimuove il segmento */
2         break;
3       case SHM_LOCK: /* Esegui il lock sul segmento */
4         break;
5       case SHM_UNLOCK: /* Esegui unlock sul segmento */
6         break;
7       default: /* Comando sconosciuto passato a do_shmctl */
8         break;
9     }
10    /* La funzione do_shmctl esegue il comando scelto dall'utente */
11    do_shmctl(shmid, cmd, &shmid_ds);
12    exit(0);
13  }
14
15  void do_shmctl(int shmid, int cmd, struct shmid_ds* buf) {
16    int rtrn; /* valore di ritorno della shmctl */
17
18    fprintf(stderr,"shmctl: Chiamo shmctl(%d, %d, buf)\n", shmid, cmd);
```

Memoria Condivisa: Esempio (Controller - part 5)

```
1   if (cmd == IPC_SET) {
2     fprintf(stderr, "\tbuf->shm_perm.uid == %d\n",
3                   buf->shm_perm.uid);
4     fprintf(stderr, "\tbuf->shm_perm.gid == %d\n",
5                   buf->shm_perm.gid);
6     fprintf(stderr, "\tbuf->shm_perm.mode == %#o\n",
7                   buf->shm_perm.mode);
8   }
9   if ((rtrn = shmctl(shmid, cmd, buf)) == -1) {
10    perror("shmctl: shmctl fallita.");
11    exit(1);
12  } else {
13    fprintf(stderr, "shmctl: shmctl ha ritornato %d\n", rtrn);
14  }
15  if (cmd != IPC_STAT && cmd != IPC_SET)
16    return; /* ritorno perche' il comando e' stato eseguito e non
17                  devo visualizzare nessuna informazione sullo stato */
```

Memoria Condivisa: Esempio (Controller - part 6)

```
1   /* Stampa lo stato corrente del segmento */
2   fprintf(stderr, "\nCurrent status:\n");
3   fprintf(stderr, "\tshm_perm.uid = %d\n", buf->shm_perm.uid);
4   fprintf(stderr, "\tshm_perm.gid = %d\n", buf->shm_perm.gid);
5   fprintf(stderr, "\tshm_perm.cuid = %d\n", buf->shm_perm.cuid);
6   fprintf(stderr, "\tshm_perm.cgid = %d\n", buf->shm_perm.cgid);
7   fprintf(stderr, "\tshm_perm.mode = %#o\n", buf->shm_perm.mode);
8   fprintf(stderr, "\tshm_perm.key = %#x\n", buf->shm_perm.__key);
9   fprintf(stderr, "\tshm_segsz = %d\n", buf->shm_segsz);
10  fprintf(stderr, "\tshm_lpid = %d\n", buf->shm_lpid);
11  fprintf(stderr, "\tshm_cpid = %d\n", buf->shm_cpid);
12  fprintf(stderr, "\tshm_nattch = %d\n", buf->shm_nattch);
13  if (buf->shm_atime)
14    fprintf(stderr, "\tshm_atime = %s", ctime(&buf->shm_atime));
15  if (buf->shm_dtime)
16    fprintf(stderr, "\tshm_dtime = %s", ctime(&buf->shm_dtime));
17  fprintf(stderr, "\tshm_ctime = %s", ctime(&buf->shm_ctime));
18 }
```

Memoria Condivisa: Esempio (Shm1 - part 1)

```
1  /*************************************************************
2  NOME: shm1.c
3  SCOPO: ``attaccare'' due volte un'area di memoria condivisa
4         Ricordarsi di rimuovere la memoria condivisa al termine
5         del programma lanciando shmctl.c oppure tramite il comando
6         della shell ipcrm.
7  *************************************************************/
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <sys/types.h>
11 #include <sys/ipc.h>
12 #include <sys/shm.h>
13 #define K 1
14 #define SHMKEY 75
15 #define N 10
```

Memoria Condivisa: Esempio (Shm1 - part 2)

```
1  int shmid;
2  int main (int argc, char *argv[]) {
3    int i, *pint;
4    char *addr1, *addr2;
5
6    /* Creao il segmento condiviso di dimensione 128*K byte */
7    shmid = shmget(SHMKEY, 128*K, 0777|IPC_CREAT);
8    /* Attacco il segmento in due zone diverse */
9    addr1 = shmat(shmid,0,0);
10   addr2 = shmat(shmid,0,0);
11
12   printf("Addr1 = 0x%x\t Address2 = 0x%x\t\n", addr1,addr2);
```

Memoria Condivisa: Esempio (Shm1 - part 3)

```
1     /* scrivo nella regione 1 */
2     pint = (int*)addr1;
3     for (i=0;i<N;i++) {
4       *pint = i;
5       printf("Writing: Index %4d\tValue: %4d\tAddress: 0x%x\n",
6                           i,*pint,pint);
7       pint++;
8     }
9     /* leggo dalla regione 2 */
10    pint = (int*)addr2;
11    for (i=0;i<N;i++) {
12      printf("Reading: Index %4d\tValue: %4d\tAddress: 0x%x\n",
13                          i,*pint,pint);
14      pint++;
15    }
16  }
```

Memoria Condivisa: Esempio (Shm2 - part 1)

```
1  /*************************************************************
2  NOME:   shm2.c
3  SCOPO:  ``attaccarsi'' ad un area di memoria condivisa
4  USO:    lanciare prima il programma shm1.c per creare la memoria
5          condivisa.
6          Ricordarsi di rimuovere la memoria condivisa al termine
7          del programma lanciando shmctl.c oppure tramite il comando
8          della shell ipcrm.
9  *************************************************************/
10 #include <sys/types.h>
11 #include <sys/ipc.h>
12 #include <sys/shm.h>
13 #include <stdio.h>
14 #define K 1
15 #define N 20
16 #define SHMKEY 75
17
18 int shmid;
```

Memoria Condivisa: Esempio (Shm2 - part 2)

```
1  int main (int argc, char *argv[]) {
2    int i, *pint;
3    char *addr;
4
5    /*mi attacco alla regione creata dal programma shm1.c*/
6    shmid = shmget(SHMKEY, 128*K, 0777);
7    addr = shmat(shmid,0,0);
8    printf("Address = 0x%x\n", addr);
9    pint = (int*) addr;
10   /* leggo dalla regione attaccata in precedenza */
11   for (i=0;i<N;i++) {
12     printf("Reading: (Value = %4d)\n",*pint++);
13   }
14 }
```

Memoria Condivisa: Esempio (Server - part 1)

```
1  /*************************************************************
2  MODULO: shm_server.c
3  SCOPO: server memoria condivisa
4  USO: lanciare il programma shm_server.c in una shell
5       e il programma shm_client.c in un'altra shell
6       Ricordarsi di rimuovere la memoria condivisa creata
7       dal server al termine del programma lanciando shmctl.c
8       oppure tramite il comando della shell ipcrm.
9  *************************************************************/
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <sys/types.h>
13 #include <sys/ipc.h>
14 #include <sys/shm.h>
15 #include <stdio.h>
16 #define SHMSZ 27
```

Memoria Condivisa: Esempio (Server - part 2)

```
1  int main(int argc, char *argv[]) {
2    char c;
3    int shmid;
4    key_t key;
5    char *shm, *s;
6    key = 5678;
7
8
9    /* Creo il segmento */
10   if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
11     perror("shmget");
12     exit(1);
13   }
14   /* Attacco il segmento all'area dati del processo */
15   if ((shm = shmat(shmid, NULL, 0)) == (void *) -1) {
16     perror("shmat");
17     exit(1);
18   }
```

Memoria Condivisa: Esempio (Server - part 3)

```
1     s = shm;
2     for (c = 'a'; c <= 'z'; c++)
3       *s++ = c;
4
5     *s = NULL;
6     while (*shm != '*')
7       sleep(1);
8
9     printf("Received '*'. Exiting...\n");
10    exit(0);
11  }
```

Memoria Condivisa: Esempio (Client - part 1)

```
1  /*************************************************************
2  MODULO: shm_client.c
3  SCOPO: client memoria condivisa
4  USO: lanciare il programma shm_server.c in una shell
5       e il programma shm_client.c in un'altra shell
6       Ricordarsi di rimuovere la memoria condivisa creata
7       dal server al termine del programma lanciando shmctl.c
8       oppure tramite il comando della shell ipcrm.
9  *************************************************************/
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <sys/types.h>
13 #include <sys/ipc.h>
14 #include <sys/shm.h>
15 #include <stdio.h>
16 #define SHMSZ 27
```

Memoria Condivisa: Esempio (Client - part 2)

```
1  int main(int argc, char *argv[]) {
2    int shmid;
3    key_t key;
4    char *shm, *s;
5    key = 5678;
6
7    /* Recupero il segmento creato dal server */
8    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
9      perror("shmget");
10     exit(1);
11   }
12
13   /* Attacco il segmento all'area dati del processo */
14   if ((shm = shmat(shmid, NULL, 0)) == (void *) -1) {
15     perror("shmat");
16     exit(1);
17   }
```

Memoria Condivisa: Esempio (Client - part 3)

```
1    /* Stampo il contenuto della memoria */
2    printf("Contenuto del segmento condiviso con il server:");
3    for (s = shm; *s != NULL; s++)
4      putchar(*s);
5
6    putchar('\n');
7    sleep(3);
8    /* ritorno * al server affinche' possa terminare */
9    *shm = '*';
10   exit(0);
11 }
```

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

- I semafori sono differenti dalle altre forme di IPC.
- Sono contatori usati per controllare l'accesso a risorse condivise da processi diversi.
- Il protocollo per accedere alla risorsa è il seguente
  1. testare il semaforo;
  2. se > 0, allora si può usare la risorsa (e viene decrementato il semaforo);
  3. se $== 0$, processo va in *sleep* finchè il semaforo non ridiventa > 0, a quel punto *wake up* e goto step 1;
- Quando ha terminato con la risorsa, incrementa il semaforo.
- Se il semaforo è a 0 significa che si sta utilizzando il 100% della risorsa.
- Un semaforo binario (valori possibili 0 e 1) è detto *mutex*.

Semafori - (cont.)

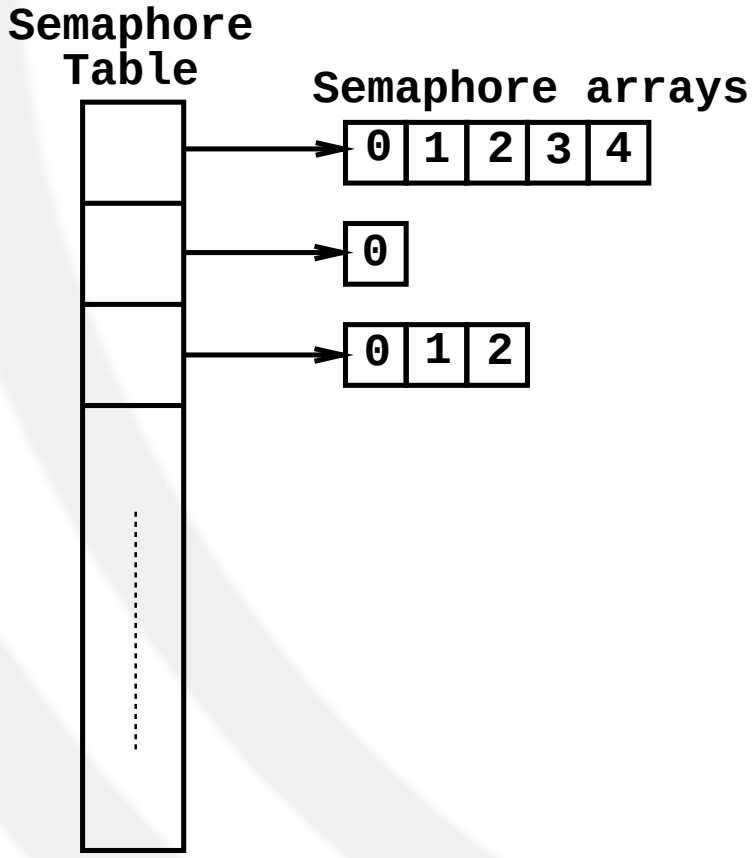
- È importante notare che per implementare correttamente un semaforo l'operazione di verifica del valore del semaforo ed il decremento/incremento devono costituire una operazione atomica.
- Per questa ragione i semafori sono implementati all'interno del kernel.

Semafori - (cont.)

- Il semaforo binario è la forma più semplice.
  - Controlla un'unica risorsa ed il suo valore è inizializzato a 1.
  - E.g., stampante capace di gestire 1 sola stampa alla volta.
- Forma più complessa.
  - Inizializzato ad un valore positivo indicante il numero di risorse che sono a disposizione per essere condivise.
  - E.g., stampante capace di gestire 10 stampe alla volta.

Semafori (System V 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 semget(key_t key, int nsems, int semflg);
```

- I valori di `key` e di `semflg` sono identici al caso delle code di messaggi e della shared memory
- `nsems` è il numero di semafori identificati dal `semid` (quanti semafori sono contenuti nel vettore).
  - Il primo semaforo del vettore ha indice 0
- Esempi
  - `semid = semget(key, 1, IPC_CREAT|IPC_EXCL|0666);`
    crea un insieme con un solo semaforo
  - `semid = semget(key, 0, 0666);`
    recupera un semaforo esistente
- NOTA: anche se Linux inizializza i semafori a zero, una applicazione portabile non deve fare affidamento su questo comportamento.

Operazioni su Semafori

```
/* 4o parametro opzionale di tipo semun (args) */
int semctl(int semid, int semnum, int cmd, ...);
union semun {
  int val; /* SETVAL */
  struct semid_ds* buffer; /* IPC_STAT, IPC_SET */
  unsigned short *array; /* GET_ALL, SET_ALL */
  struct seminfo *__buf; /* IPC_INFO solo per Linux */
};
```

- Esegue l'operazione di controllo specificata da *cmd* sull'insieme di semafori specificato da *semid* o sul *semnum*-esimo semaforo dell'insieme. A seconda del comando il quarto parametro è opzionale.
- Operazioni (`cmd`):

`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 semafori 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

- `buffer` è un puntatore ad una struttura `semid_ds` definita in `sys/sem.h`:

```
struct semid_ds   {
    struct ipc_perm sem_perm;  /* operation permission struct */
    time_t sem_otime;          /* Last semop time */
    time_t sem_ctime;          /* Last change time */
    unsigned short sem_nsems;  /* No. of semaphores */
  };
```

- La union `semun` deve essere definita nel codice del processo che chiama la `semctl()`
- `ipc_perm` è una struttura definita in `sys/ipc.h`:

```
struct ipc_perm {
  key_t          __key; /* Key supplied to semget(2) */
  uid_t          uid;   /* Effective UID of owner */
  gid_t          gid;   /* Effective GID of owner */
  uid_t          cuid;  /* Effective UID of creator */
  gid_t          cgid;  /* Effective GID of creator */
  unsigned short mode;  /* Permissions */
  unsigned short __seq; /* Sequence number */
};
```

Operazioni su Semafori

La `semctl()` ritorna valori diversi a seconda del comando eseguito :

- `GETNCNT`: il numero di processi in attesa che il semaforo `semnum` venga incrementato
- `GETPID`: il PID dell'ultimo processo che ha effettuato `semop()` sul semaforo `semnum` dell'insieme
- `GETVAL`: il valore del semaforo `semnum` dell'insieme
- `GETZCNT`: il numero di processi in attesa che il semaforo `semnum` diventi 0

Semafori: Esempio (Controller - part 1)

```
1  /*************************************************************
2  MODULO: semctl.c
3  SCOPO:  Illustrare il funz. di semctl()
4  USO:    prima di lanciare semctl.c() creare un semaforo usando un
5          altro programma (ad esempio sem.x)
6  *************************************************************/
7  #include   <stdlib.h>
8  #include   <stdio.h>
9  #include   <sys/types.h>
10 #include   <sys/ipc.h>
11 #include   <sys/sem.h>
12 #include   <time.h>
13
14 struct semid_ds semid_ds;
```

Semafori: Esempio (Controller - part 2)

```
1  /* explicit declaration required */
2  union semun {
3    int val;
4    struct semid_ds* buf;
5    unsigned short int *array;
6    struct seminfo *__buf;
7  } arg;
8
9  void do_semctl(int semid, int semnum, int cmd, union semun arg);
10 void do_stat();
11 char warning_message[] = "If you remove read permission for yourself,
12                           this program will fail frequently!";
13
14 int main(int argc, char *argv[]) {
15   union semun arg; /* union to pass to semctl() */
16   int cmd;         /* command to give to semctl() */
17   int i;
18   int semid;       /* semid to pass to semctl() */
19   int semnum;      /* semnum to pass to semctl() */
```

Semafori: Esempio (Controller - part 3)

```
1   fprintf(stderr, "Enter semid value: ");
2   scanf("%i", &semid);
3
4   fprintf(stderr, "Valid semctl cmd values are:\n");
5   fprintf(stderr, "\tGETALL = %d\n", GETALL);
6   fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
7   fprintf(stderr, "\tGETPID = %d\n", GETPID);
8   fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
9   fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
10  fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
11  fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
12  fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
13  fprintf(stderr, "\tSETALL = %d\n", SETALL);
14  fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
15  fprintf(stderr, "\nEnter cmd: ");
16  scanf("%i", &cmd);
```

Semafori: Esempio (Controller - part 4)

```
1   /* Do some setup operations needed by multiple commands. */
2   switch (cmd) {
3     case GETVAL:
4     case SETVAL:
5     case GETNCNT:
6     case GETZCNT:
7     /* Get the semaphore number for these commands. */
8       fprintf(stderr, "\nEnter semnum value: ");
9       scanf("%i", &semnum);
10      break;
11    case GETALL:
12    case SETALL:
13      /* Allocate a buffer for the semaphore values. */
14      fprintf(stderr,"Get number of semaphores in the set.\n");
15      arg.buf = &semid_ds;
16      /* when IPC_STAT is called the second argument of semctl()
17         is ignored. IPC_STAT is called to retrieve info semid_ds
18         on the semaphore set */
19      do_semctl(semid, 0, IPC_STAT, arg);
20      if (arg.array = (u_short *)malloc(
21                (unsigned) (semid_ds.sem_nsems * sizeof(u_short)))) {
22         /* Break out if you got what you needed */
23         break;
24      }
```

Semafori: Esempio (Controller - part 5)

```
1       fprintf(stderr,"semctl: unable to allocate spacefor %d values\n",
2               semid_ds.sem_nsems);
3       exit(2);
4   }
5
6   /*Get the rest of the arguments needed for the specified command.*/
7   switch (cmd) {
8     case SETVAL:
9       /* Set value of one semaphore. */
10      fprintf(stderr, "\nEnter semaphore value: ");
11      scanf("%i", &arg.val);
12      do_semctl(semid, semnum, SETVAL, arg);
13      /* Fall through to verify the result. */
14      fprintf(stderr,
15          "Executing semctl GETVAL command to verify results...\n");
16    case GETVAL:
17      /* Get value of one semaphore. */
18      arg.val = 0;
19      do_semctl(semid, semnum, GETVAL, arg);
20      break;
```

Semafori: Esempio (Controller - part 6)

```
1     case GETPID:
2       /* Get PID of the last process that successfully completes a
3       semctl(SETVAL), semctl(SETALL), or semop() on the semaphore. */
4       arg.val = 0;
5       do_semctl(semid, 0, GETPID, arg);
6       break;
7     case GETNCNT:
8       /* Get number of processes waiting for semaphore value
9          to increase. */
10      arg.val = 0;
11      do_semctl(semid, semnum, GETNCNT, arg);
12      break;
13    case GETZCNT:
14      /* Get number of processes waiting for semaphore value to
15         become zero. */
16      arg.val = 0;
17      do_semctl(semid, semnum, GETZCNT, arg);
18      break;
```

Semafori: Esempio (Controller - part 7)

```
1     case SETALL:
2       /* Set the values of all semaphores in the set. */
3       fprintf(stderr, "There are %d semaphores in the set.\n",
4           semid_ds.sem_nsems);
5       fprintf(stderr, "Enter semaphore values:\n");
6       for (i = 0; i < semid_ds.sem_nsems; i++) {
7         fprintf(stderr, "Semaphore %d: ", i);
8         scanf("%hi", &arg.array[i]);
9       }
10      do_semctl(semid, 0, SETALL, arg);
11      /* Fall through to verify the results. */
12      fprintf(stderr,"Do semctl GETALL command to verify results.\n");
13    case GETALL:
14      /* Get and print the values of all semaphores in the set.*/
15      do_semctl(semid, 0, GETALL, arg);
16      fprintf(stderr, "The values of the %d semaphores are:\n",
17          semid_ds.sem_nsems);
18      for (i = 0; i < semid_ds.sem_nsems; i++)
19        fprintf(stderr, "%d ", arg.array[i]);
20      fprintf(stderr, "\n");
21      break;
```

Semafori: Esempio (Controller - part 8)

```
1     case IPC_SET:
2       /* Modify mode and/or ownership. */
3       arg.buf = &semid_ds;
4       do_semctl(semid, 0, IPC_STAT, arg);
5       fprintf(stderr, "Status before IPC_SET:\n");
6       do_stat();
7       fprintf(stderr, "Enter sem_perm.uid value: ");
8       scanf("%hi", &semid_ds.sem_perm.uid);
9       fprintf(stderr, "Enter sem_perm.gid value: ");
10      scanf("%hi", &semid_ds.sem_perm.gid);
11      fprintf(stderr, "%s\n", warning_message);
12      fprintf(stderr, "Enter sem_perm.mode value: ");
13      scanf("%hi", &semid_ds.sem_perm.mode);
14      do_semctl(semid, 0, IPC_SET, arg);
15      /* Fall through to verify changes. */
16      fprintf(stderr, "Status after IPC_SET:\n");
17    case IPC_STAT:
18      /* Get and print current status. */
19      arg.buf = &semid_ds;
20      do_semctl(semid, 0, IPC_STAT, arg);
21      do_stat();
22      break;
```

Semafori: Esempio (Controller - part 9)

```
1       case IPC_RMID:
2         /* Remove the semaphore set. */
3         arg.val = 0;
4         do_semctl(semid, 0, IPC_RMID, arg);
5         break;
6       default:
7         /* Pass unknown command to semctl. */
8         arg.val = 0;
9         do_semctl(semid, 0, cmd, arg);
10        break;
11    }
12  exit(0);
13 }
14
15 void do_semctl(int semid, int semnum, int cmd, union semun arg) {
16   int i;
17
18   fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ",
19         semid, semnum, cmd);
```

Semafori: Esempio (Controller - part 10)

```
1  void do_semctl(int semid, int semnum, int cmd, union semun arg) {
2    int i;
3
4    fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ",
5        semid, semnum, cmd);
6    switch (cmd) {
7      case GETALL:
8        fprintf(stderr, "arg.array = %#x)\n", arg.array);
9        break;
10     case IPC_STAT:
11     case IPC_SET:
12       fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
13       break;
14     case SETALL:
15       fprintf(stderr, "arg.array = [");
16       for (i = 0;i < semid_ds.sem_nsems;i++) {
17         fprintf(stderr, "%d", arg.array[i]);
18        if (i < semid_ds.sem_nsems)
19           fprintf(stderr, ", ");
20       }
21       fprintf(stderr, "])\n");
22       break;
23   }
```

Semafori: Esempio (Controller - part 11)

```
1    /* call to semctl() */
2    i = semctl(semid, semnum, cmd, arg);
3    if (i == -1) {
4      perror("semctl: semctl failed");
5      exit(1);
6    }
7    fprintf(stderr, "semctl: semctl returned %d\n", i);
8    return;
9  }
10
11 void do_stat() {
12   fprintf(stderr, "sem_perm.uid = %d\n", semid_ds.sem_perm.uid);
13   fprintf(stderr, "sem_perm.gid = %d\n", semid_ds.sem_perm.gid);
14   fprintf(stderr, "sem_perm.cuid = %d\n", semid_ds.sem_perm.cuid);
15   fprintf(stderr, "sem_perm.cgid = %d\n", semid_ds.sem_perm.cgid);
16   fprintf(stderr, "sem_perm.mode = %#o, ", semid_ds.sem_perm.mode);
17   fprintf(stderr, "access permissions = %#o\n",
18       semid_ds.sem_perm.mode & 0777);
19   fprintf(stderr, "sem_nsems = %d\n", semid_ds.sem_nsems);
20   fprintf(stderr, "sem_otime = %s",
21       semid_ds.sem_otime ? ctime(&semid_ds.sem_otime) : "Not Set\n");
22   fprintf(stderr, "sem_ctime = %s", ctime(&semid_ds.sem_ctime));
23 }
```

Operazioni su Semafori

```
int semop(int semid, struct sembuf* sops, unsigned nsops);
```

- Applica l'insieme (array) `sops` di operazioni (in numero pari a `nsops`) all'insieme di semafori `semid`.
- Le operazioni, contenute in un array 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 - |sem_op| < 0` altrimenti 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)

Operazioni su Semafori

NOTA BENE: L'insieme di operazioni inserite in una chiamata alla system call `semop()` viene eseguito in modo atomico. Se una delle operazioni non può essere eseguita, il comportamento della system call `semop()` dipende dalla flag `IPC_NOWAIT`:

- se `IPC_NOWAIT` è settato, `semop()` fallisce e ritorna -1;
- se `IPC_NOWAIT` non è settato, il processo viene bloccato;

Semafori: Esempio (sem - part 1)

```
1  /*************************************************
2  MODULO: sem.c
3  SCOPO:  Illustrare il funz. di semop() e semctl()
4  USO:    Lanciare il programma sem.x e quindi
5          semop.x o semctl.x in una shell separata
6  *************************************************/
7
8  #include     <stdio.h>
9  #include     <stdlib.h>
10 #include     <sys/types.h>
11 #include     <sys/ipc.h>
12 #include     <sys/sem.h>
13
14 #define KEY 14
```

Semafori: Esempio (sem - part 2)

```
1  int main(int argc, char *argv[]) {
2
3    int semid;  /* identificatore dei semafori */
4    int i;
5    /* struttura per le operazioni sui semafori */
6    struct sembuf * sops =
7              (struct sembuf *) malloc (sizeof(struct sembuf));
8
9    /*
10     Creazione di due semafori con permessi di lettura e scrittura
11     per tutti. Le flag ICP_CREAT e IPC_EXCL fanno si che la
12     funzione semget ritorni errore se esiste già un vettore di
13     semafori con chiave KEY. Vedere man semget.
14   */
15   if((semid = semget(KEY, 2, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
16     perror("semget");
17     exit(1);
18   }
19
20   /* Inizializzo i due semafori a 1 */
```

Semafori: Esempio (sem - part 3)

```
1   /*
2     Per eseguire operazioni ATOMICHE sui semafori si usa la funzione
3     semop(). Vedere man semop.
4     Alla funzione semop() vengono passati 3 argomenti:
5     - l'identificatore dell'array di semafori su cui eseguire
6     l'operazione
7     - il puntatore alla struttura sembuf necessaria per eseguire
8     le operazioni
9     - il numero di operazioni da eseguire
10
11    Per ogni operazione da eseguire è necessario creare una
12    struttura di tipo sembuf. La struttura contiene 3 campi:
13    - il numero del semaforo da utilizzare. Ricordare che la semget
14    ritorna array di semafori.
15    - un intero N che rappresenta l'operazione da eseguire.
16      Se l'intero N e' > 0 il valore del semaforo viene incrementato
17      di tale quantità. Se N = 0 la semop blocca il processo in
18      attesa che il valore del semaforo diventi 0. Se N < 0 la semop
19      blocca il processo in attesa che il valore del semaforo meno N
20      sia maggiore o uguale a 0.
21    - una eventuale flag (IPC_NOWAIT o SEM_UNDO)
22      IPC_NOWAIT serve per avere semafori non bloccanti
23      SEM_UNDO serve per ripristinare il vecchio valore del semaforo
24                in caso di terminazioni precoci.
25  */
```

Semafori: Esempio (sem - part 4)

```
1     sops->sem_num = 0;   /* semaforo 0 */
2     sops->sem_op = 1;    /* incrementa di uno il valore del
3                             semaforo 0 */
4     sops->sem_flg = 0;   /* nessuna flag settata */
5     /* esecuzione dell'operazione sul semaforo 0 */
6     semop(semid, sops, 1);
7
8     sops->sem_num = 1;   /* semaforo 1 */
9     sops->sem_op = 1;    /* incrementa di uno il valore del
10                            semaforo 1 */
11    sops->sem_flg = 0;
12    /* esecuzione dell'operazione sul semaforo 1 */
13    semop(semid, sops, 1);
14
15    printf("I semafori sono stati inizializzati a 1.\n");
16    printf("Lanciare il programma semctl.x o semop.x su un'altra
17           shell e fornire semid=%d\n", semid);
```

Semafori: Esempio (sem - part 5)

```
1       while(1) {
2
3         sops->sem_num = 0;   /* semaforo 0 */
4         sops->sem_op = 0;    /* attende che il semaforo valga zero */
5         sops->sem_flg = 0;
6         /* quando viene eseguita questa operazione il codice si
7            blocca in attesa che il valore del semaforo 0 diventi 0 */
8         semop(semid, sops, 1);
9         /* Quando il semaforo diventa 0 stampo che il processo
10           e' stato sbloccato */
11        printf("Sbloccato 1\n");
12
13        sops->sem_num = 1;   /* semaforo 1 */
14        sops->sem_op = 0;    /* attende che il semaforo valga zero */
15        sops->sem_flg = 0;
16
17        /* quando viene eseguita questa operazione il codice
18         si blocca in attesa che il valore del semaforo 1 diventi 0 */
19        semop(semid, sops, 1);
20        /* Quando il semaforo diventa 0 stampo che il processo
21           e' stato sbloccato */
22        printf("                Sbloccato 2\n");
23
24      }
25  }
```

Semafori: Esempio (semop - part 1)

```
 1 /*****************************************
 2 MODULO: semop.c
 3 SCOPO: Illustrare il funz. di semop()
 4 *****************************************/
 5 #include     <stdio.h>
 6 #include     <stdlib.h>
 7 #include     <sys/types.h>
 8 #include     <sys/ipc.h>
 9 #include     <sys/sem.h>
10
11 union semun {
12   int               val;    /* Valore per SETVAL */
13   struct semid_ds  *buf;    /* Buffer per IPC_STAT, IPC_SET */
14   unsigned short   *array;  /* Array per GETALL, SETALL */
15   struct seminfo   *__buf;  /* Buffer per IPC_INFO (specifico
16                                di Linux) */
17 };
18
19 int ask(int* semidp, struct sembuf **sopsp);
```

Semafori: Esempio (semop - part 2)

```
1  static struct semid_ds  semid_ds;  /* stato del set di semafori */
2  static char error_mesg1[] = "semop: Non posso allocare spazio per un
3                               vettore di %d valori.\n";
4  static char error_mesg2[] = "semop: Non posso allocare spazio per %d
5                               strutture sembuf. \n";
6
7  int main(int argc, char* argv[]) {
8    int     i;
9    int     nsops;          /* numero di operazioni da fare */
10   int     semid;          /* semid per il set di semafori */
11   struct sembuf *sops; /* puntatore alle operazioni da eseguire */
12
13   /* Cicla finche' l'utente vuole eseguire operazioni chiamando la
14      funzione ask */
15   while (nsops = ask(&semid, &sops)) {
16     /* Inizializza il vettore di operazioni da eseguire */
17     for (i = 0; i < nsops; i++) {
18       fprintf(stderr,
19       "\nInserire il valore per l'operazione %d di %d.\n",i+1,nsops);
20       fprintf(stderr,"sem_num(i valori validi sono 0<=sem_num<%d): ",
21                semid_ds.sem_nsems);
22       scanf("%d", &sops[i].sem_num);
```

Semafori: Esempio (semop - part 3)

```
1       fprintf(stderr, "sem_op: ");
2       scanf("%d", &sops[i].sem_op);
3       fprintf(stderr,"Possibili flag per sem_flg:\n");
4       fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",IPC_NOWAIT);
5       fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",SEM_UNDO);
6       fprintf(stderr, "\tNESSUNO =\t%6d\n",0);
7       fprintf(stderr, "sem_flg: ");
8       /* controllare cosa fa %i su man scanf */
9       scanf("%i", &sops[i].sem_flg);
10    }
11
12    /* Ricapitola la chiamata da fare a semop() */
13    fprintf(stderr,"\nsemop: Chiamo semop(%d, &sops, %d) with:",
14            semid, nsops);
15    for (i = 0; i < nsops; i++) {
16      fprintf(stderr, "\nsops[%d].sem_num = %d, ",i,sops[i].sem_num);
17      fprintf(stderr, "sem_op = %d, ", sops[i].sem_op);
18      fprintf(stderr, "sem_flg = %o\n",sops[i].sem_flg);
19    }
```

Semafori: Esempio (semop - part 4)

```
1       /* Chiama la semop() e riporta il risultato */
2       if ((i = semop(semid, sops, nsops)) == -1) {
3         perror("semop: semop failed");
4       } else {
5         fprintf(stderr, "semop: valore di ritorno = %d\n", i);
6       }
7     }
8   }
9
10
11  int ask(int *semidp, struct sembuf **sopsp) {
12    static union semun arg;      /* argomento per semctl() */
13    static int nsops = 0;        /* dimensione del vettore di sembuf */
14    static int semid = -1;       /* semid del set di semafori */
15    static struct sembuf *sops; /* puntatore al vettore di sembuf */
16    int i;
17
18    if (semid < 0) {
19      /* Prima chiamata alla funzione ask()
20         Recuperiamo semid dall'utente e lo stato corrente del set di
21         semafori */
22      fprintf(stderr,
23              "Inserire semid del set di semafori su cui operare: ");
24      scanf("%d", &semid);
```

Semafori: Esempio (semop - part 5)

```
1     *semidp = semid;
2
3     arg.buf = &semid_ds;
4
5     /* chiamata a semctl() */
6     if (semctl(semid, 0, IPC_STAT, arg) == -1) {
7       perror("semop: semctl(IPC_STAT) fallita.");
8       /* Notare che se semctl() fallisce, semid_ds viene riempita
9          con 0, e successivi test per controllare il numero di
10         semafori ritorneranno 0.
11         Se invece semctl() va a buon fine, arg.buf verra'
12         riempito con le informazioni relative al set di semafori */
13
14      /* allocazione della memoria per un vettore di interi la cui
15         dimensione dipende dal numero di semafori inclusi nel set */
16    } else if ((arg.array = (ushort *)malloc(
17                    sizeof(ushort) * semid_ds.sem_nsems)) == NULL) {
18      fprintf(stderr, error_mesg1, semid_ds.sem_nsems);
19      exit(1);
20    }
21  }
```

Semafori: Esempio (semop - part 6)

```
1   /* Stampa i valori correnti dei semafori */
2   if (semid_ds.sem_nsems != 0) {
3     fprintf(stderr, "Ci sono %d semaphores.\n", semid_ds.sem_nsems);
4     /* Chiama la funzione semctl per recuperare i valori
5        di tutti i semafori del set. Nel caso di GETALL il secondo
6        argomento della semctl() viene ignorato e si utilizza il
7        campo array della union semun arg */
8     if (semctl(semid, 0, GETALL, arg) == -1) {
9       perror("semop: semctl(GETALL) fallita");
10    } else {
11      fprintf(stderr, "I valori correnti dei semafori sono:");
12      for (i = 0; i < semid_ds.sem_nsems; i++)
13        fprintf(stderr, " %d", arg.array[i]);
14      fprintf(stderr, "\n");
15    }
16  }
```

Semafori: Esempio (semop - part 7)

```
1   /* Allocazione dello spazio per le operazioni che l'utente
2      desidera eseguire */
3   fprintf(stderr,
4       "Quante operazioni vuoi eseguire con la prossima semop()?\n");
5   fprintf(stderr, "Inserire 0 or control-D per uscire: ");
6   i = 0;
7   if (scanf("%d", &i) == EOF || i == 0)
8     exit(0);
9   if (i > nsops) {
10    if (nsops != 0) /* libero la memoria precedentemente allocata */
11      free((char *)sops);
12    nsops = i;
13    /* Allocazione della memoria per le operazioni da eseguire*/
14    if ((sops = (struct sembuf *)malloc(
15                        (nsops * sizeof(struct sembuf)))) == NULL) {
16      fprintf(stderr, error_mesg2, nsops);
17      exit(2);
18    }
19  }
20  *sopsp = sops;
21  return (i);
22 }
```