

Laboratorio Sistemi Operativi

Giovanni Tosini

Indice

1	Processi e programmi	1
2	System call	3
2.1	Gestione degli errori delle System Call	3
3	Kernel data types	7
4	Filesystem	9
4.1	File	9
4.1.1	open	9
4.1.2	read	10
4.1.3	write	11
4.1.4	lseek	12
4.1.5	close	13
4.1.6	unlink	13
4.1.7	stat, lstat, fstat	13
4.1.8	Mode	14
4.1.9	access	14
4.1.10	chmod e fchmod	15
4.2	Directory	15
4.2.1	mkdir	15
4.2.2	rmdir	15
4.2.3	opendir, closedir e readdir	16
5	Processi	19
5.1	Identificatori	19
5.1.1	getpid	19
5.1.2	getuid, geteuid, getgid e getegid	19
5.2	Environment	19
5.2.1	getenv, setenv, unsetenv	20
5.3	Working directory	20

5.3.1	getcwd	20
5.3.2	chdir, fchdir	21
5.4	File descriptor table	21
5.4.1	dup	21
5.5	Operazioni con i processi	21
5.5.1	exit, atexit	21
5.6	Creazione dei processi	22
5.6.1	fork	22
5.6.2	getppid	23
5.7	Monitoring	23
5.7.1	wait	24
5.7.2	waitpid	24
5.8	Program execution	26
5.8.1	exec system calls	26
6	Interprocess communication	29
6.1	Creazione delle chiavi	29
6.2	Data Structure	30
6.3	IPC commands	30
6.3.1	ipcs	30
6.3.2	ipcrm	31
6.4	Semaphores	31
6.4.1	semget	31
6.4.2	semctl	31
6.4.3	semop	34
6.5	Signals	35
6.5.1	Segnali di default	35
6.5.2	signal handler	36
6.5.3	signal	37
6.5.4	pause	38
6.5.5	sleep	39
6.5.6	kill	39
6.5.7	alarm	40
6.5.8	sigemptyset/sigfillset	40
6.5.9	sigaddset, sigdelset, sigismember	40
6.5.10	sigprocmask	41
7	Domande fatte all'orale	43

Capitolo 1

Processi e programmi

Un processo è un'istanza di un programma eseguito. Come viene creato, il **Kernel** gli associa una certa struttura di memoria.

Program code: segmento in sola lettura contenente istruzioni in linguaggio macchina;

Initialized data: segmento contenente variabili globali e statiche;

Uninitialized data: segmento contenente variabili globali e statiche **non** inizializzate;

Heap: segmento contenente variabili allocate dinamicamente;

Stack: segmento contenente gli argomenti e le variabili interne delle funzioni.

Una delle strutture dati di supporto è il **file descriptor table**, conterrà tutti i file che il processo aprirà. Ogni processo contiene già 3 **file descriptor** associati ad esso:

1. Standard input
2. Standard output
3. Standard error

Ogni successivo file aperto verrà identificato con il valore minore disponibile. Il file descriptor table è visibile **solo** a runtime.

Capitolo 2

System call

Sono un punto di ingresso verso il Kernel, vengono utilizzate per richiedere dei servizi. Dallo User Level verranno fatte delle chiamate alla System Call Interface che a sua volta comunicherà al Kernel.

2.1 Gestione degli errori delle System Call

Nella sezione ERRORS del comando `man` si possono trovare tutti i possibili valori di ritorno di errore di una System Call. Tuttavia è possibile usare la variabile `errno` accessibile tramite l'uso della libreria `<errno.h>`. Ci permetterà di sapere l'errore effettivo causato in base al valore salvato al suo interno.

Esempio:

```
1      #include <errno.h>
2      ...
3      //system call to open a file
4      fd = open(pathname, flags, mode);
5      //Begin code handling errors
6      if(fd == -1){
7          if(errno == EACCES){
8              //Handling not allowed access to the file
9          }
10         else{
11             //Some other error occurred
12         }
13     }
14     //End code handling errors
15     ...
16
```

La maggior parte delle system call ritorna un -1 o NULL Pointer in caso di errore, alcune però usano il -1 come valore di ritorno anche in caso di non errore. Qui l'uso di `errno` acquista ulteriore valore. Esempio:

```

1      #include <sys/resource.h>
2      ...
3      //Reset the errno variable to 0
4      errno = 0;
5      //System call getpriority gets the nice value of a
process
6      nice = getpriority(which, who);
7      if((nice == -1) && (errno != 0)){
8          //Handling getpriority errors
9      }
10     ...
11
```

Esistono altre funzioni che aiutano a gestire gli errori, come la funzione `perror()` che stampa su standard error la stringa che le viene fornita. Esempio:

```

1      #include <stdio.h>
2      ...
3      //System call to open a file
4      fd = open(pathname, flags, mode);
5      if(fd == -1){
6          perror("<Open>");
7          //System call to kill the current process
8          exit(EXIT_FAILURE);
9      }
10     ...
11
```

L'output sarà:

```

1      <Open>: No such file or directory
2
```

La libreria `string.h` fornisce la funzione `strerror()` che prende in input il valore di `errno` e stampa l'errore effettivo. Esempio:

```

1      #include <stdio.h>
2      ...
3      //System call to open a file
4      fd = open(path, flags, mode);
5      if(fd == -1){
6          printf("Error opening (%s): \n\t%s\n", path,
strerror(errno));
7          //System call to kill the current process
8          exit(EXIT_FAILURE);
9      }

```



```
10     ...  
11
```

L'output sarà il seguente:

```
1      Error opening (myFile.txt):  
2          No such file or directory  
3
```


Capitolo 3

Kernel data types

Sono delle `typedef` di tipi normali C, necessari per ovviare problemi di portabilità, per esempio il `pid_t` usando per identificare il process ID di un processo non è altro che un tipo definito come `typedef int pid_t`, quindi un intero.

Capitolo 4

Filesystem

4.1 File

4.1.1 open

Apri un file esistente e nel caso in cui non esistesse lo può creare tramite l'uso di specifiche flag, in caso di successo ritorna un file descriptor, quindi va aggiunta una riga alla file descriptor table. In caso di errore ritorna un -1.

```
1      #include <sys/stat.h>
2      #include <stdio.h>
3
4      //Returns file descriptor on successo, or -1 on error
5      int open(const char *pathname, int flags, .../*mode_t
6      mode*/);
```

- il `pathname` può essere il nome del file o il suo eventuale path;
- la flag può essere un bit mask di una o più flag che definiscono l'accesso al file, possono essere ORate fra di loro tramite "|";
- le mode possono si comportano in maniera simile alle flag, definiscono i permessi che il file avrà.

Tabella con le flag disponibili:

Flag	Description
O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for reading and writing
O_TRUNC	Truncate existing file to zero length
O_APPEND	Writes are always appended to end of file
O_CREAT	Create file if it doesn't already exist
O_EXCL	With O_CREAT, ensure that this call creates the file.

Tabella delle mode disponibili:

Flag	Description
S_IRWXU	user has read, write, and execute permission
S_IRUSR	user has read permission
S_IWUSR	user has write permission
S_IXUSR	user has execute permission
S_IRWXG	group has read, write, and execute permission
S_IRGRP	group has read permission
S_IWGRP	group has write permission
S_IXGRP	group has execute permission
S_IRWXO	others has read, write, and execute permission
S_IROTH	others has read permission
S_IWOTH	others has write permission
S_IXOTH	others has execute permission

Se non vengono forniti i permessi cosa succederà al file? All'interno del SO esiste la `umask` con dei valori che di default non dà permessi allo user e solo scrittura a group e others, tale valore sarà 022. Di `umask` ne esiste una sola, andando a fornire dei permessi tramite la `open` i permessi che il file avrà saranno la `mode` con il negato della `umask` (`mode and ~umask`). Vari esempi di utilizzo:

```

1      int fd;
2      //Open existing file for only writing
3      fd = open("myfile", O_WRONLY);
4
5      //Open new or existing file for reading/writing,
6      truncating
7      // to zero bytes; file permissions read+write only
8      for owner
9      fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC,
10     S_IRUSR | S_IWUSR);

```

4.1.2 read

Prende in input il file descriptor ottenuto tramite la `open`, un `buffer` dove andremo a salvare quello che leggeremo dal file e un `size_t` che definisce il numero di byte che vogliamo leggere dal file. In caso di successo ritornerà un valore `ssize_t` che dovrebbe essere uguale o minore a `count`, in di errore tornerà un -1.

```

1      #include <stdio.h>
2
3      //Returns number of bytes read, or -1 on error
4      ssize_t read(int fd, void *buf, size_t count);
5

```

Esempio d'uso:

```

1      //Open existing file for reading
2      int fd = open("myfile", O_RDONLY);
3      if(fd == -1)
4          errExit("open");
5
6      // A MAX_READ bytes buffer
7      char buffer[MAX_READ + 1];
8
9      //Reading up to MAX_READ bytes from myfile
10     ssize_t numRead = read(fd, buffer, MAX_READ);
11     if(numRead == -1)
12         errExit("Read");
13

```

Un esempio di lettura da Standard Input:

```

1      // A MAX_READ bytes buffer
2      char buffer[MAX_READ + 1];
3
4      //Reading up to MAX_READ bytes from STDIN
5      ssize_t numRead = read(STDIN_FILENO, buffer, MAX_READ
6      );
7      if(numRead == -1)
8          errExit("read");
9
10     buffer[numRead] = '\0';
11     printf("Input data: %s\n", buffer);

```

4.1.3 write

Ci permette di scrivere su un file descriptor

```

1      #include <unistd.h>
2
3      //Returns number of bytes written, or -1 on error
4      ssize_t write(int fd, void *buf, size_t count);
5

```

Esempio di scrittura:

```

1      //Open existing file for writing
2      int fd = open("myfile", O_WRONLY);
3      if(fd == -1)
4          errExit("open");
5
6      //A buffer collecting the string
7      char buffer[] = "Ciao Mondo";
8
9      //Writing up to sizeof(buffer) bytes into myfile

```

```

10     ssize_t numWrite = write(fd, buffer, sizeof(buffer));
11     if(numWrite != sizeof(buffer))
12         errExit("write");
13

```

Per scrivere su terminale, come prima si userà `STDOUT_FILENO` al posto del file descriptor.

4.1.4 lseek

Una volta aperto un file, il kernel salva un file offset ovvero un indicatore un valore che identifica a quale punto di scrittura/lettura siamo arrivati. Per utilizzare tale cursore useremo la `lseek`.

```

1     #include <unistd.h>
2
3     //Returns the resulting offset location, or -1 on
4     error
5     off_t write(int fd, off_t offset, int whence);

```

N.B.: whence indica la base di partenza dell'offset;

Esempio di utilizzo:

```

1     #include <unistd.h>
2
3     //Returns number of bytes written, or -1 on error
4     ssize_t write(int fd, void *buf, size_t count);
5

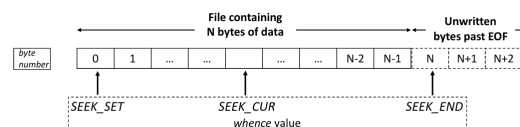
```

Alcuni esempi:

```

1     //first byte of the file
2     off_t current = lseek(fd1, 0, SEEK_SET);
3     //last byte of the file
4     off_t current = lseek(fd2, -1, SEEK_END);
5     //10th byte past the current offset location of the
6     file
7     off_t current = lseek(fd3, -10, SEEK_CUR);
8     //10th byte after the current offset location of the
9     file
10    off_t current = lseek(fd4, 10, SEEK_CUR);

```



4.1.5 close

```
1      #include <unistd.h>
2
3      //Returns 0 on success, or -1 on error
4      int close(int fd);
5
```

Tutti i file descriptor vengono chiusi quando un processo termina, ma è buona prassi chiudere sempre. La chiusura **non** elimina il file.

4.1.6 unlink

```
1      #include <unistd.h>
2
3      //Returns 0 on success, or -1 on error
4      int unlink(const char *pathname);
5
```

Prende in input il nome del file, perché il file descriptor può anche essere chiuso, se il file non ha altri symbolic link, viene rimosso.

Symbolic link: il collegamento su desktop, oppure il file è aperto da altri processi.

unlink **non** può rimuovere directory.

4.1.7 stat, lstat, fstat

```
1      #include <sys/stat.h>
2
3      //Returns 0 on success, or -1 on error
4      int stat(const char *pathname, struct stat *statbuf);
5      int lstat(const char *pathname, struct stat *statbuf)
6      ;
7      int fstat(int fd, struct stat *statbuf);
```

In caso di successo la **struct stat** viene popolata da varie informazioni. La differenza tra queste system call sono:

- **stat** ritorna informazioni relative a un file tramite il nome o path;
- **lstat** tramite symbolic link;
- **fstat** utilizza il file descriptor;

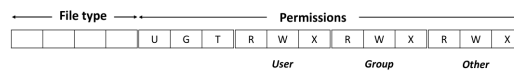
4.1.8 Mode

Si tratta di una bit mask, presente anche nella `struct stat`, che ci permette di definire i permessi dei file. Lunga 16 bit, i primi 9 sono per other, group e user, rispettivamente 3 a testa. Possiamo usarla per avere informazioni sulla tipologia del file. Esempio:

```

1  char pathname[] = "/tmp/file.txt";
2  struct stat statbuf;
3  //Getting the attribute of /tmp/file.txt
4  if(stat(pathname, &statbuf) == -1)
5      errExit("stat");
6
7  //Checking if /tmp/file.txt is a regular file
8  if((statbuf.st_mode & S_IFMT) == S_IFREG)
9      printf("regular file!\n");
10
11  //Equivalently, checking if /tmp/file.txt is a
12  //regular file by S_ISREG macro
13  if(S_ISREG(statbuf.st_mode))
14      printf("regular file!\n");
15

```



I bit oltre i 9 dedicati a other, group e user hanno i seguenti significati:

- U identifica se l'utente che sta eseguendo quell'eseguibile è lo stesso utente proprietario dell'eseguibile;
- G verifica se il gruppo che sta eseguendo è il gruppo proprietario;
- T è lo sticky bit, funziona come un bit che non ci permette di cancellare quel file;

File e directory sono la stessa cosa, per modificare i permessi di una directory si usano le stesse mode dei file.

4.1.9 access

Controlla l'accessibilità di un file relativamente al nostro user id e group id.

```

1  #include <unistd.h>
2
3  //Returns 0 if all permissions are granted, otherwise
4  -1
5  int access(const char *pathname, int mode)

```

Le possibili mode che si possono usare insieme a questa system call sono queste:

Constant	Description
F_OK	Does the file exist?
R_OK	Can the file be read?
W_OK	Can the file be written?
X_OK	Can the file be executed?

4.1.10 chmod e fchmod

Permettono di cambiare i permessi di un file prendendo in input il pathname o il file descriptor più le eventuali mode di interesse.

```

1 //All return 0 on success, or -1 on error
2 #include <sys/stat.h>
3
4 int chmod(const char *pathname, mode_t mode);
5
6 #define _BSD_SOURCE
7 #include <sys/stat.h>
8
9 int fchmod(int fd, mode_t mode);
10
```

4.2 Directory

4.2.1 mkdir

Prende in input il pathname della directory e una mode.

```

1 #include <sys/stat.h>
2
3 //Returns 0 on success, or -1 on error.
4 int mkdir(const char *pathname, mode_t mode);
5
```

I parametri mode sono gli stessi della `open`, se la directory esistesse già, il valore di ritorno sarà sempre -1, ma la variabile `errno` conterrà il messaggio `EEXIST`, a indicare che tale directory è già presente.

4.2.2 rmdir

```

1      #include <unistd.h>
2
3      //Returns 0 on success, or -1 on error.
4      int rmdir(const char *pathname);
5

```

Se esiste anche un solo file all'interno dello directory tale system call tornerà -1 di errore, per avere successo deve essere completamente vuota.

4.2.3 opendir, closedir e readdir

```

1      #include <sys/types.h>
2      #include <dirent.h>
3
4      //Returns directory stream handler, or NULL on error.
5      DIR *opendir(const char *dirpath);
6
7      //Returns 0 on success, or -1 on error
8      int closedir(DIR *dirp);
9

```

Una volta aperta una directory per leggerla si userà

```

1      #include <sys/types.h>
2      #include <dirent.h>
3
4      //Returns pointer to an allocated structure
5      //describing the
6      //next directory entry or NULL on end-of-directory or
7      //error
8      struct dirent *readdir(DIR *dirp);
9

```

La struttura di riferimento per la readdir:

```

struct dirent {
    ino_t      d_ino;      /* Inode number */
    off_t      d_off;      /* Not an offset; see below */
    unsigned short d_reclen; /* Length of this record */
    unsigned char d_type;    /* Type of file; not supported
                             * by all filesystem types */
    char        d_name[256]; /* Null-terminated filename */
};

```

Tramite il `d_type` possiamo ottenere delle informazioni sul tipo di file che stiamo scorrendo:

Constant	File type
DT_BLK	block device
DT_CHR	character device
DT_DIR	directory
DT_FIFO	named pipe (FIFO)
DT_LNK	symbolic link
DT_REG	regular file
DT SOCK	UNIX socket

Esempio d'uso:

```
1  DIR *dp = opendir("myDir");
2  if(dp == NULL)
3      return -1;
4
5  errno = 0;
6  struct dirent *dentry;
7  //Iterate until NULL is returned as a result
8  while((dentry = readdir(dp)) != NULL){
9      if(dentry->d_type == DT_REG)
10         printf("Regular file: %s\n", dentry->d_name);
11         errno = 0;
12     }
13     //NULL is returned on error, and when the end-of-
14     directory is reached!
15     if(errno != 0)
16         printf("Error while reading dir.\n");
17     closedir(dp);
```


Capitolo 5

Processi

5.1 Identificatori

Ogni processo è caratterizzato da un PID univoco che cambia sempre, l'unico processo a mantenere sempre lo stesso identificatore è il processo `INIT`.

5.1.1 `getpid`

```
1  #include <unistd.h>
2  #include <sys/types.h>
3
4  pid_t getpid(void);
5
```

Ritorna come valore il PID del processo chiamante e **non può fallire**.

5.1.2 `getuid`, `geteuid`, `getgid` e `getegid`

Anche queste system call **hanno sempre successo**. La differenza `getuid` e `geteuid`, ovvero tra real user ed effective user, consiste che il real user (vale anche per il real group), identificano l'utente o il gruppo a cui appartiene il processo, mentre l'effective è quello che viene usato dal SO, per gestire operazioni solitamente non permesse (come installare tramite packet manager, si usa `sudo`).

5.2 Environment

Ad ogni processo viene associato un array `environ`, di stringhe che contiene tutte le variabili di ambiente salvate all'interno di `environ`

Esempio:

```

1      #include <stdio.h>
2      //Global variable pointing to the environment of the
   process
3      extern char **environ;
4
5      int main(int argc, char *argv[]){
6          for(char **it = environ; (*it) != NULL; ++it){
7              printf("--> %s\n", *it);
8          }
9          return 0;
10     }
11

```

5.2.1 getenv, setenv, unsetenv

```

1      #include <stdlib.h>
2      //Returns pointer to (value) string, or NULL if no
   such variable exists
3      char *getenv(const char *name);
4      //Returns 0 on success, or -1 on error
5      int setenv(const char *name, const char *value, int
   overwrite);
6      //Returns 0 on success, or -1 on error
7      int unsetenv(const char *name);
8

```

Sono system call che interagiscono con l'environment.

5.3 Working directory

5.3.1 getcwd

Per identificare la directory in cui io sto lavorando in questo momento, si usa la system call `getcwd`

```

1      #include <unistd.h>
2
3      //Returns cwdbuf on success, or NULL on error
4      char *getcwd(char *cwdbuf, size_t size);
5

```

Ritorna NULL nell'eventualità che il pathname sia più lungo della `size` data.

5.3.2 chdir, fchdir

```
1      #include <unistd.h>
2
3      //Returns 0 on success, or -1 on error
4      int chdir(const char *pathname);
5
```

Permette di cambiare la directory attuale tramite l'uso del pathname.

```
1      #define _BSD_SOURCE
2      #include <unistd.h>
3
4      //Returns 0 on success, or -1 on error
5      int fchdir(int fd);
6
```

Permette di cambiare la directory attuale tramite l'uso del file descriptor.

5.4 File descriptor table

Per visualizzare la file descriptor table di un processo basta andare all'interno della cartella `/proc/<PID>/fd`, dove `fd` è un symbolic link per ogni riga della tabella. Si potranno trovare anche socket e pipe.

5.4.1 dup

```
1      #include <unistd.h>
2
3      //Returns (new) file descriptor on success, or -1 on
4      error
5      int dup(int oldfd);
```

Ritornerà un nuovo file descriptor a partire da uno che si ha già, ovviamente partendo dal valore più basso disponibile.

5.5 Operazioni con i processi

5.5.1 exit, atexit

```
1      #include <stdlib.h> //N.B. provided by C library
2
3      void exit(int status);
4
```

Al suo interno chiama un'altra system call `_exit`, va **sempre** a buon fine. La terminazione del processo può essere gestita da noi tramite `atexit`

```

1      #include <stdlib.h>
2      //Returns 0 on success, or nonzero on error
3
4      int atexit(void (*func)(void));
5

```

Il puntatore a funzione preso come parametro, quando viene creato non va a finire all'interno del layout di memoria del processo.

Esempio pratico

```

1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <unistd.h>
4
5      void func1(){
6          printf("\tAtexit function 1 called\n");
7      }
8      void func2(){
9          printf("\tAtexit function 2 called\n");
10     }
11
12     int main(int argc, char *argv[]){
13         if(atexit(func1) != 0 || atexit(func2) != 0)
14             _exit(EXIT_FAILURE);
15         exit(EXIT_SUCCESS);
16     }
17

```

L'eventuale output sarà

```

1      Atexit function 2 called
2      Atexit function 1 called
3

```

Eseguire un `return(n)` è equivalente all'eseguire un `exit(n)`, se il `return` non viene messo il programma metterà in automatico il `return(0)`.

5.6 Creazione dei processi

5.6.1 fork

```

1      #include <unistd.h>
2
3      //In parent: returns process ID of child on success,
4      //or -1 on error
5
6      //In created child: always returns 0
7

```

```
5     pid_t fork(void);
6
```

Il processo figlio sarà una copia del padre in tutto e per tutto e inizieranno a eseguire in parallelo, ovviamente l'esecuzione non è sincrona. L'unica differenza tra i due è la variabile di ritorno `pid_t` che sarà 0 per il figlio, mentre per il padre conterrà il PID del figlio. Si userà il valore 0 per far fare delle operazioni specifiche esclusivamente il figlio magari.

Il figlio eredita tutto quello che il padre aveva già aperto, istanziato, etc. Esempio:

```
1     #include <unistd.h>
2
3     int main(){
4         int stack = 111;
5         pid_t pid = fork();
6         if(pid == -1)
7             errExit("fork");
8         //Both parent and child come here
9         if(pid == 0)
10             stack = stack *4;
11         printf("\t%s stack %d\n", (pid == 0) ? "(child)"
12 : "(parent)", stack);
13     }
```

5.6.2 getppid

Permette di ottenere il pid del padre.

```
1     #include <unistd.h>
2
3     //Always succesfully returns PID of caller's parent
4     pid_t getppid(void);
5
```

Torna sempre il PID del padre di norma, ma se dovesse succedere che il padre termini prima del figlio, in quel caso tornerà il PID del processo che ha ereditato il figlio, di normale tale processo è INIT ovvero il processo con PID 1.

5.7 Monitoring

I metodi del padre per monitorare i figli

5.7.1 wait

```
1      #include <sys/wait.h>
2
3      //Returns PID of terminated child, or -1 on error
4      pid_t wait(int *status);
5
```

Prende in input uno status o anche NULL, con NULL come status, il padre aspetterà la terminazione di uno qualunque dei suoi figli. Questa system call blocca il padre.

Se un padre che non ha più figli, fa la `wait` gli ritornerà un -1, ma per capire che non ci sono più figli ovviamente dobbiamo guardare il contenuto della variabile `errno` che in questa casistica conterrà `ECHILD`.

Se si volesse aspettare tutti i figli occorrerà mettere questa chiamata in un ciclo `while`, se status non è NULL la system call darà come ritorno gli stati di terminazione del figlio come `exit(1)` o `exit(0)`.

5.7.2 waitpid

La differenza con la `wait` è che questa aspetta un figlio specifico in base al valore del PID

```
1      #include <sys/wait.h>
2
3      //Returns a PID, 0 or -1 on error
4      pid_t waitpid(pid_t pid, int *status, int options);
5
```

In base al valore inserito in `pid` si comporta diversamente:

- `pid ≥ 0`, aspetta il figlio con quello specifico PID;
- `pid = 0`, aspetta la terminazione di ogni figlio nello stesso process group (originati tutti dallo stesso padre);
- `pid < -1`, aspetta che uno dei processi del process group con quel PID passato, termini;
- `pid = -1`, aspetto indistintamente che ogni processo termini.

Per le options si possono usare:

WUNTRACED : ritorna il PID del figlio sia quando viene terminato che quando viene stoppato;

WCONTINUED : ritorna quando un figlio è stato rimesso in esecuzione dopo essere stato stoppato;

WNOHANG : non è bloccante, quindi fino a quando i figli indicati dal pid non hanno cambiato status, il padre che la chiama continuerà a fare altro, in questo caso il valore di ritorno della `waitpid` è 0;

0 : aspetta solo per i figli che terminano.

```

1      pid_t pid;
2      for(int i = 0; i < 3; ++i){
3          pid = fork();
4          if(pid == 0){
5              //Code executed by the child process...
6              exit(0);
7          }
8      }
9      //The parent process only waits for the last created
10     child
11     waitpid(pid, NULL, 0);

```

Altro esempio:

```

1      pid_t pid = fork();
2      if(pid == 0){
3          //Code executed by the child process
4      }
5      else{
6          //Waiting for a terminated/stopped | resumed
7      child process
8          waitpid(pid, NULL, WUNTRACED | WCONTINUED);
9      }

```

Lo status è un intero a 16 bit, gli 8 bit più significativi vengono usati per capire lo status di uscita(quindi i possibili valori vanno da 0 a 255). Inoltre vengono fornite dal SO delle macro per capire come il processo figlio ha terminato

WIFEXITED : ritorna true se il figlio termina normalmente;

WEXITSTATUS : ritorna lo status con cui ha terminato il processo figlio;

WIFSIGNALED : ritorna true se il processo figlio è stato ucciso con un segnale;

WTERMSIG : ritorna il numero del segnale che ha causato la terminazione del figlio;

WIFSTOPPED : ritorna true se il processo è stato stoppato con un segnale;

WSTOPSIG : ritorna il numero del segnale che stoppato il processo figlio;

WIFCONTINUED : ritorna true se il figlio ha ripreso l'esecuzione tramite un SIGCONT.

Esempi vari:

```

1      waitpid(-1, &status, WUNTRACED | WCONTINUED);
2      if(WIFEXITED(status)){
3          printf("Child exited, status = %s\n", WEXITSTATUS
4              (status));
5      }

```

```

1      waitpid(-1, &status, WUNTRACED | WCONTINUED);
2      if(WIFSIGNALED(status)){
3          printf("child killed by signal %d (%s)", WTERMSIG
4              (status), strsignal(WTERMSIG(status)));
5      }

```

5.8 Program execution

5.8.1 exec system calls

La system call **exec**, non crea figli, un processo che chiama tale system call viene rimodellato, prende l'eseguibile a cui punta e lo rimappa all'interno del processo chiamante. Trasforma tutto il contenuto del processo chiamante, il PID **non** cambia.

```

1      #include <unistd.h>
2      //None of the following returns on success, all
3      return -1 on error
4      int execl(const char *path, const char *arg, ...); //
5      variadic functions
6      int execlp(const char *path, const char *arg, ...);
7      int execln(const char *path, const char *arg, ...,
8          char *const envp[]);
9      int execv(const char *path, char *const argv[]);
10     int execvp(const char *path, char *const argv[]);
11     int execve(const char *path, char *const argv[], char
12         *const envp[]);

```

L'ultimo parametro delle **execl** deve essere sempre un NULL.

function	path	arguments (argv)	environment (envp)
<code>execl</code>	pathname	list	caller's environ
<code>execlp</code>	filename	list	caller's environ
<code>execle</code>	pathname	list	array
<code>execv</code>	pathname	array	caller's environ
<code>execvp</code>	filename	array	caller's environ
<code>execve</code>	pathname	array	array

path : per pathname ci si riferisce al path assoluto all'eseguibile, mentre con filename al nome dell'eseguibile che si deve trovare nella lista delle directory del PATH environ;

argv : una lista o array terminata da NULL, che definiscono gli argomenti del programma;

envp : un array di puntatori a stringhe terminato da NULL che definiscono l'environ del programma.

Esempio:

```

1      #include <stdio.h>
2      #include <unistd.h>
3      #include <stdlib.h>
4
5      int main(int argc, char *argv[]){
6          printf("PID of example.c = %d\n", getpid());
7          char *args[] = {"Hello.c", "C", "Programming",
8              NULL};
9          execv("./hello", args);
10         printf("Back to example.c");
11
12         return 0;
13     }

```

Eseguendo sia questo codice che "Hello.c" il primo programma andrà a essere rimodellato con il codice all'interno di "Hello.c".

Capitolo 6

Interprocess communication

Sono dei meccanismi utilizzati per coordinare attività tra vari processi, in parole povere servono a sincronizzare i processi. Di base le IPC possono essere create tramite un comando con `get`, prendono come primo parametro una chiave che serve ai processi per poter comunicare tra di loro, senza non riuscirebbero. Ognuna di queste ritorna un file descriptor.

6.1 Creazione delle chiavi

Le chiavi delle system V IPC, sono dei tipi `key_t` e praticamente sono degli `int`, possono essere definite da noi o lasciare che si arrangi il SO. Una chiave è univoca un eventuale creazione di due IPC differenti con la stessa chiave porterebbe a una sovrascrittura. Per delegarne la creazione al SO si utilizza la macro `IPC_PRIVATE`, che in automatico genera una chiave univoca verificando prima le chiavi già in uso.

Un'altra modalità è tramite l'uso della system call `ftok`

```
1      #include <sys/ipc.h>
2
3      //Returns integer key on success, or -1 on error(
4      check_errno)
5      key_t ftok(char *pathname, int proj_id);
```

Il `proj_id` può essere un qualunque `int` l'importante che i suoi ultimi 8 bit siano diversi da 0 visto che saranno quelli che verranno usati per la creazione, usando dei caratteri ASCII si ha la certezza che gli ultimi 8 bit non saranno uguali a 0.

Un ultimo possibile metodo è definirle manualmente, con il rischio di usare chiavi già in uso.

6.2 Data Structure

Ogni IPC ha una sua struttura, ma ognuna ha in comune un'unica struttura, la `ipc_perm`:

```

1      struct ipc_perm{
2          key_t __key;           //Key, as supplied to '
get' call
3          uid_t uid;             //Owner's user ID
4          gid_t gid;             //Owner's group ID
5          uid_t cuid;            //Creator's user ID
6          gid_t cgid;            //Creator's group ID
7          unsigned short mode;    //Permissions
8          unsigned short __seq;   //Sequence number
9      }
10

```

La `mode` sono una serie di permessi che possono essere settati. `cuid` e `gid` sono immutabili. Inoltre le IPC possono avere solo permessi di lettura e scrittura, non di esecuzione. Esempio:

```

1      struct semid_ds semq;
2      //get the data structure of a semaphore from the
kernel
3      if(semctl(semid, 0, IPC_STAT, &semq) == -1)
4          errExit("semctl get failed");
5      //change the owner of the semaphore
6      semq.sem_perm.uid = newuid;
7      //update the kernel copy of the data structure
8      if(semctl(semid, IPC_SET, &semq) == -1)
9          errExit("semctl set failed");
10

```

6.3 IPC commands

6.3.1 ipcs

`ipcs` da terminale, mostra tutti i dettagli delle IPC esistenti in quell'istante all'interno del SO. Tali dettagli sono:

- key;
- owner;
- permessi;
- dimensione;

Più altri dettagli specifici per la singola IPC.

6.3.2 ipcrm

`ipcrm` permette di eliminare delle IPC che sono ancora attive, nel caso in cui non venga gestita la cosa dal programmatore.

6.4 Semaphores

6.4.1 semget

`semget` è la system call che permette di creare un set di semafori.

```
1      #include <sys/sem.h>
2
3      //Returns semaphore set identifier on success, or -1
on error
4      int semget(key_t key, int nsems, int semflg);
5
```

Prende come parametri una chiave, un numero di semafori che dovrà essere contenuto dal set di semafori e infine prende input delle flag. Tali flag possono essere una qualsiasi della `open` (non i permessi di esecuzione siccome le IPC non ne necessitano), in aggiunta si metteranno sempre in or `IPC_CREAT` e la `IPC_EXCL`, che differenza c'è tra queste due flag? La prima dice che se il semaforo con quella chiave, non esiste lo crea, nel caso in cui esista, restituisce l'ID di quel set di semafori. La seconda insieme alla prima farà fallire la system call in caso la chiave in input sia già in uso per un altro set di semafori. Questo è comodo in base al fatto che si voglia creare un set di semafori nuovo oppure ottenere l'ID di un set esistente.

6.4.2 semctl

Permette di svolgere svariate operazioni su un determinato set di semafori.

```
1      #include <sys/sem.h>
2
3      //Returns non negative integer on success, or -1 on
error
4      int semctl(int semid, int semnum, int cmd, .../*union
semnum arg*/);
5
```

`semid` è l'identificatore del set di semafori, su cui le operazioni verranno effettuate. `semnum` è un identificatore specifico per un semaforo all'interno del set (vuoi lavoro con il terzo semaforo di un set compost da 10? metti 3), mettendo 0 si agirà sull'intero set. `cmd` possono essere una serie di flag. Una `union` funziona in maniera simile a una `struct`, permette di definire al

suo interno vari campi, ma dal punto di vista del SO, per quella `union` verrà allocata una memoria pari alla dimensione della variabile più grande al suo interno.

Esempio:

```

1      #ifndef SEMUN_H
2      #define SEMUN_H
3      #include <sys/sem.h>
4      //definition of the union semun
5      union semun{
6          int val;                //per operazioni su
singolo semaforo
7          struct semid_ds *buf;   //su permessi
8          unsigned short *array; //su un intero set di
semafori
9      };
10     #endif
11
```

In questo caso verrà allocata memoria equivalente alla `struct semid_ds`.

Flag

```

1      int semctl(semid, 0 /*ignored*/, cmd, arg);
2
```

L'elenco delle flag da usare con `semctl` al parametro `cmd`

- `IPC_RMID`: deve essere usata per eliminare l'IPC alla fine del suo utilizzo;
- `IPC_STAT`: permette di salvare lo stato dell'IPC all'interno della `struct semid_ds`, verrà salvato all'interno di `arg.buf`;
- `IPC_SET`: occorre per settare i nuovi permessi passando in input `arg` modificato;

```

1      struct semid_ds{
2          struct ipc_perm sem_perm; //ownership and
permissions
3          time_t sem_otime; //Time of last semop()
4          time_t sem_ctime; //Time of last change
5          unsigned long sem_nsems; //Number of semaphores
in set
6      };
7
```

Esempio di modifica dei permessi di un set di semafori:

```

1      int semid = semget(key, 10, IPC_CREAT | S_IRUSR |
    S_IWUSR);
2      //instantiate a semid_ds struct
3      struct semid_ds ds;
4      //instantiate a semun union(defined manually
    somewhere)
5      union semun arg;
6      arg.buf = &ds;
7      //get a copy of semid_ds structure belonging to the
    kernel
8      if(semctl(semid, 0 /*ignored*/, IPC_STAT, arg) == -1)
9          errExit("semctl IPC_STAT failed");
10     //update permissions to guarantee read access to the
    group
11     arg.buf->sem_perms.mode |= S_IRGRP;
12     //update the semid_ds structure of the kernel
13     if(semctl(semid, 0 /*ignored*/, IPC_SET, arg) == -1)
14         errExit("semctl IPC_SET failed");
15

```

Semaphore Control Operations

```

1      int semctl(semid, semnum, cmd, arg);
2

```

Per usare i cmd SETVAL e GETVAL occorre specificare il semaforo singolo del set.

- SETVAL si usa il campo val della union e si setta il singolo semaforo a quel valore;
- GETVAL ritorna il valore del semaforo. (modificato da "ritorna il valore in arg.val)

Se invece si vuole lavorare con l'intero set di semafori,

```

1      int semctl(semid, 0 /*ignored*/, cmd, arg);
2

```

- SETALL si usa il campo array della union e si setta l'intero set del semaforo a quel con quei valori;
- GETALL salva in arg.array tutti i valori dei semafori.

Esempio per settare i valori di un set:

```

1      int semid = semget(key, 10, IPC_CREAT | S_IRUSR |
2      S_IWUSR);
3      //set the first 5 semaphores to 1, and the remaining
4      to 0
5      int values[] = {1, 1, 1, 1, 1, 0, 0, 0, 0, 0};
6      union semun arg;
7      arg.array = values;
8      //initialize the semaphore set
9      if(semctl(semid, 0/*ignored*/, SETALL, arg) == -1)
10         errExit("semctl SETALL");

```

Altre operazioni:

```

1      int semctl(semid, semnum, cmd, 0);
2

```

- GETPID: ritorna il PID dell'ultimo processo che ha fatto una semop sul semaforo specificato;
- GETNCNT: ritorna il numero dei processi in attesa che il valore di un semaforo diventi positivo;
- GETZCNT: ritorna il numero di processi che attualmente stanno aspettando che un semaforo diventi zero.

6.4.3 semop

```

1      #include <sys/sem.h>
2
3      //Returns 0 on success, or -1 on error
4      int semop(int semid, struct sembuf *sops, unsigned
5      int nsops);

```

La struttura `sembuf` è la seguente:

```

1      struct sembuf{
2          unsigned shor sem_num; //semaphore number
3          short sem_op; //operation to be performed
4          short sem_flag; //operation flags
5      }
6

```

`nsops` indica la dimensione della `struct`, sono tutte operazioni a livello atomico. Se `sem_op` è positiva, si va ad aggiungere tale valore al semaforo in oggetto, di conseguenza se negativa avverrà una differenza. Invece se `sem_op`

è uguale a 0 e il semaforo è positivo, il processo che chiama rimarrà bloccato fino a quando il semaforo non sarà uguale a 0.

Usando la `sem_flag` `IPC_NOWAIT`, la chiamata non sarà bloccante e tornerà un errore `EAGAIN`.

Se il set di semafori viene rimosso, tutti i processi bloccati vengono risvegliati e avranno in output `EINTR` a indicare che lo sblocco è avvenuto per l'eliminazione del semaforo.

6.5 Signals

Un segnale è una notifica a un processo che qualcosa è avvenuto, un processo riceve un segnale nel momento in cui avviene un certo evento o un altro processo invia un segnale a un altro processo. Durante il tempo che passa tra la generazione del segnale e la ricezione si dice che il segnale è **pendente**. Normalmente un segnale pendente raggiunge il processo target nell'istante immediato in cui il target riottiene la possibilità di usare la CPU, nel caso in cui la stesse già utilizzando il segnale verrà ricevuto pressochè istantaneamente. Una volta ricevuto un segnale il processo destinatario a seconda del segnale ricevuto:

- terminerà;
- si sospenderà/bloccherà;
- verrà ripreso dopo essere stato bloccato;
- il segnale verrà ignorato, quindi scartato dal kernel senza alcun effetto sul processo;
- il processo eseguirà un **signal handler**, una funzione definita dal programmatore che esegue determinate azioni in base al segnale ricevuto.

6.5.1 Segnali di default

Segnali per terminare un processo

SIGTERM : permette di terminare un processo, può essere gestito per lasciare il sistema in uno stato "safe";

SIGINT : viene inviato a un processo quando si fa CTRL+C, anche questo è gestibile;

SIGQUIT : usato per generare dei core dump e fare debug;

SIGKILL : immediato, non è gestibile e termina il processo;

Segnali per bloccare e riprendere

SIGSTOP : blocca un processo immediatamente, non può essere bloccato, ignorato o gestito da un **handler**;

SIGCONT : fa riprendere l'esecuzione a un processo precedentemente bloccato.

Altri segnali

SIGPIPE : generato quando un processo prova a scrivere su una PIPE o FIFO per le quali non c'è il corrispondente processo lettore;

SIGALRM : segnale che usa un processo per mandare un segnale a sè stesso dopo un certo lasso di tempo;

SIGUSR1 e **SIGUSR2** : vengono usati come più pare e piace da parte del programmatore.

Nome	Numero	Può essere gestito?	Azione di default
SIGTERM	15	sì	termina un processo
SIGINT	2	sì	termina un processo
SIGQUIT	3	sì	core dump + terminazione
SIGKILL	9	no	termina un processo
SIGSTOP	17	no	blocca un processo
SIGCONT	19	sì	riprende un processo
SIGPIPE	13	sì	termina un processo
SIGALRM	14	sì	termina un processo
SIGUSR1	30	sì	termina un processo
SIGUSR2	31	sì	termina un processo

Una tabella riassuntiva, la colonna "numero" indica il codice per l'architettura x86 per inviare quel segnale da terminale tramite il comando **kill**.

6.5.2 signal handler

Un **signal handler** è una funzione chiamata come uno specifico segnale viene consegnato a un processo, la sua forma generale è la seguente:


```

1     void sigHandler(int sig){
2         //code for the handler
3     }
4

```

La funzione non ha alcun valore di ritorno, il parametro che prende è il segnale specifico. Quando il **signal handler** è chiamato dal kernel, **sig** è impostato con il valore numerico del segnale consegnato al processo.

Di base il SO ha un suo handler di default, che fa delle operazioni sui dati di un processo che riceve un **SIGSTOP** o **SIGKILL**.

La chiamata a un **signal handler** può interrompere il flusso del programma principale in qualsiasi momento. Il kernel invoca la funzione e una volta conclusa il programma riprende se può, da dove era stato interrotto.

6.5.3 signal

Una volta definito il **signal handler** la system call **signal** permette di settare quello definito al posto di quello di default.

```

1     #include <signal.>
2
3     typedef void (*signalhandler_t)(int);
4     //Returns previous signal disposition on success, or
5     SIG_ERR on error
6     signalhandler_t signal(int signum, signalhandler_t handler)
7

```

signum identifica il segnale che voglia gestire, mentre **handler** può essere:

- l'indirizzo di un **signal handler** definito dall'utente;
- la costante **SIG_DFL** che resetta la disposizione di default del processo per il segnale **signum**;
- la costante **SIG_IGN** che setta il processo per ignorare la ricezione di una segnale **signum**.

Esempio:

```

1     void sigHandler(int sig){
2         printf("The signal %s was caught!\n",
3             (sig == SIGINT) ? "CTRL-C" : "signal user-1");
4     }
5     int main(int argc, char *argv[]){
6         //setting sigHandler to be executed for SIGINT or
7         SIGUSR1
8         if(signal(SIGINT, sigHandler) == SIG_ERR ||

```

```

8         signal(SIGUSR1, sigHandler) == SIG_ERR){
9             errExit("change signal handler failed");
10        }
11        //Do something else her. During this time, if
12        SIGINT/SIGUSR1
13        //is delivered, sigHandler will be used to handle
14        the signal.
15        //Reset the default process disposition for
16        SIGINT and SIGUSR1
17        if(signal(SIGINT, SIG_DFL) == SIG_ERR ||
18        signal(SIGUSR1, SIG_DFL) == SIG_ERR){
19            errExit("reset signal handler failed");
20        }
21        return 0;
22    }

```

N.B.: la SIG_DFL è una flag specifica per resettare.

Riassunto:

- SIGKILL e SIGSTOP non possono essere gestite;
- un segnale è un evento asincrono, non possiamo predire quando arriverà;
- quando viene chiamato un `signal handler` il segnale che l'ha invocato viene automaticamente bloccato e sbloccato solo successivamente alla fine del `signal handler`;
- se un segnale bloccato viene generato più volte, una volta sbloccato verrà consegnato al processo solo una;
- l'esecuzione di un `signal handler` può essere interrotta da un segnale non bloccato;
- le disposizioni del segnale sono ereditate dai figli del processo padre.

6.5.4 pause

```

1     #include <unistd.h>
2     //Always return -1 with errno set to EINTR
3     int pause();
4

```

Sospende l'esecuzione del processo fino a quando un segnale non viene catturato dal `signal handler` o un segnale non gestito lo termina.

6.5.5 sleep

```
1      #include <unistd.h>
2      unsigned int sleep(unsigned int seconds);
3      //Returns 0 on normale completion, or
4      number of unslept seconds if prematurely terminated
5
```

Sospende l'esecuzione del processo chiamante per un numero specifico di secondi specificato nel parametro `seconds` o fino a quando un segnale non viene catturato interrompendo la sospensione.

6.5.6 kill

```
1      #include <signal.h>
2
3      //Returns 0 on success, or -1 on error
4      int kill(pid_t pid, int sig);
5
```

Questa system call prende in input il PID a del processo a cui si vuole inviare il segnale e il tipo di segnale che si vuole inviare.

- (PID > 0): il segnale verrà inviato al processo con quel PID specifico;
- (PID = 0): il segnale viene inviato a tutti i processo appartenenti allo stesso gruppo, incluso il processo chiamante;
- (PID < 0): il segnale viene mandato a tutti i processi di un certo gruppo che ha un ID equivalente al PID in assoluto;
- (PID = -1): il segnale viene inviato a tutti i processi tranne INIT, se stesso o quelli a cui non si può inviare un segnale.

Esempio:

```
1      int main(int argc, char *argv[]){
2          pid_t child = fork();
3          switch(child){
4              case -1:
5                  errExit("fork");
6              case 0: //Child process
7                  while(1);
8              default: //Parent process
9                  sleep(10); //wait 10 seconds
10                 kill(child, SIGKILL); //kill the child
11
12     process
13     }
```

```
12         return 0;
13     }
14
```

6.5.7 alarm

```
1     #include <signal.h>
2
3     //Always succeeds, returning number of seconds
    remaining on
4     //any previously set timer, or 0 if no timer
    previously was set
5     unsigned int alarm(unsigned int seconds);
6
```

Invia il segnale SIGALRM al processo chiamante dopo un periodo preciso di secondi. Se viene richiamata dopo va a resettare i secondi.

6.5.8 sigemptyset/sigfillset

```
1     #include <signal.h>
2
3     typedef unsigned long sigset_t;
4
5     //Both return 0 on success, or -1 on error
6     int sigemptyset(sigset_t *set);
7     int sigfillset(sigset_t *set);
8
```

`sigset_t` è un tipo di dato che rappresenta l'insieme dei segnali. `sigemptyset` e `sigfillset` devono essere usate per inizializzare il set dei segnali prima di usarlo in qualsiasi altra maniera. La prima inizializza il set contenendo alcun segnale, mentre la seconda lo inizializza per contenerli tutti.

6.5.9 sigaddset, sigdelset, sigismember

Dopo l'inizializzazione, si possono aggiungere o rimuovere segnali individuali dall'insieme tramite `sigaddset` e `sigdelset`:

```
1     #include <signal.h>
2
3     //Both return 0 on success, or -1 on error
4     int sigaddset(sigset_t *set, int sig);
5     int sigdelset(sigset_t *set, int sig);
6
```

Si può anche verificare se uno specifico segnale sia presente all'interno del set:

```
1      #include <signal.h>
2
3      //Returns 1 if sig is member of set, otherwise 0
4      int sigismember(const sigset_t *set, int sig);
5
```

6.5.10 sigprocmask

```
1      #include <signal.h>
2
3      //Returns 0 on success, or -1 on error
4      int sigprocmask(int how, const sigset_t *set,
5      sigset_t *oldset);
```

Per ogni processo il kernel ha una maschera dei segnali, ovvero un insieme di segnali che non possono essere inviati al processo. Se si prova a inviare un segnale appartenente alla maschera, al processo, la consegna di tale segnale verrà posticipata fino a quando non sarà sbloccato il segnale, rimuovendolo dalla maschera. Il parametro `how` può assumere vari valori:

- `SIG_BLOCK`, il set dei segnali bloccati sarà l'unione tra il set corrente e quello nuovo;
- `SIG_UNBLOCK`, prende tutti i segnali passati l'argomento `set` e li toglie dalla maschera corrente;
- `SIG_SETMASK`, setta il `set` passato, come maschera del processo.

Se l'argomento `oldset` non è `NULL`, punterà a un buffer in cui verrà salvata la precedente maschera. Se invece si vuole recuperare la maschera senza cambiarla, si specifica `NULL` al posto di `set`, in quel caso `how` sarà ignorato.

Capitolo 7

Domande fatte all'orale

1. **D:** Dove va a finire l'handler dell'exit? (lezione 2 al 1:00:00)

R: L'handler viene preso dal SO così com'è e viene mappato da qualche parte all'interno del SO, non all'interno della memoria del processo. Quando viene chiamato si va fuori e poi si ritorna all'interno del processo.

2. **D:** Se il figlio ha terminato e il padre non riesce a eseguire la `wait`, cosa succede?

R: Quel figlio viene trasformato in un processo di cui manteniamo non tutto il layout di memoria, ma solo informazioni base come il PID, lo status di terminazione e le risorse usate. Viene trasformato in un processo zombie. Quando il padre esegue la `wait` vede che il figlio ha già terminato e allora il SO rimuove il processo zombie. La `wait` ha questo scopo di tenere il sistema più flessibile possibile, INIT tramite la `wait` eredita i processi zombie ed eventualmente li termina.

3. **D:** Differenza tra `exec` e `fork`

4. **D:** Mostra come posso modificare i permessi di un determinato semaforo.

R: serve l'ID del semaphore set, tramite la `semctl` passo tale valore con `IPC_STAT` che verrà salvato in `arg.buf`, in cui posso entrare dentro e da lì modificare i permessi.

5. **D:** Dove avviene l'handling da parte del signal handler?

R: Avviene all'interno del Kernel.