Laboratorio Sistemi Operativi

Giovanni Tosini

Indice

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: segmente 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` accedibile 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 occured
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:

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

```

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:

```
        #include <stdio.h>
        ...
        //System call to open a file
        fd = open(pathname, flags, mode);
        if(fd == -1){
            perror("<Open>");
            //System call to kill the current process
            exit(EXIT_FAILURE);
        }
        ...

```

L'output sarà:

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

```

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

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

```
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

Apre 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 success, or -1 on error
5        int open(const char *pathname, int flags, .../*mode_t
     mode*/);
6
```

- 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,
   truncating
6       // to zero bytes; file permissions read+write only
   for owner
7       fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC,
   S_IRUSR | S_IWUSR);
8
```

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:

```
        //Open existing file for reading
        int fd = open("myfile", O_RDONLY);
        if(fd == -1)
            errExit("open");

        // A MAX_READ bytes buffer
        char buffer[MAX_READ + 1];

        //Reading up to MAX_READ bytes from myfile
        ssize_t numRead = read(fd, buffer, MAX_READ);
        if(numRead == -1)
            errExit("Read");

```

Un esempio di lettura da Standard Input:

```
        // A MAX_READ bytes buffer
        char buffer[MAX_READ + 1];

        //Reading up to MAX_READ bytes from STDIN
        ssize_t numRead = read(STDIN_FILENO, buffer, MAX_READ
    );
        if(numRead == -1)
            errExit("read");

        buffer[numRead] = '\0';
        printf("Input data: %s\n", buffer);

```

4.1.3 write

Ci permette di scrivere su un file descriptor

```
        #include <unistd.h>

        //Returns number of bytes written, or -1 on error
        sszie_t write(int fd, void *buf, size_t count);

```

Esempio di scrittura:

```
        //Open existing file fro writing
        int fd = open("myfile", O_WRONLY);
        int(fd == -1)
            errExit("open");

        //A buffer collecting the string
        char buffer[] = "Ciao Mondo";

        //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
    error
4           off_t write(int fd, off_t offset, int whence);
5
```

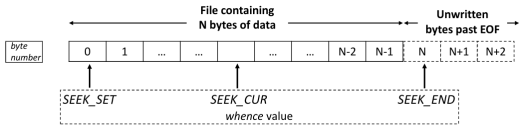
N.B.: `whence` indica la base di partenza dell'offset;

Esempio di utilizzo:

```
1           #include <unistd.h>
2
3           //Returns number of bytes written, or -1 on error
4           sszie_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
    file
6           off_t current = lseek(fd3, -10, SEEK_CUR);
7           //10th byte after the current offset location of the
    file
8           off_t current = lseek(fd4, 10, SEEK_CUR);
9
```



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         it 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         int fstat(int fd, struct stat *statbuf);
7
```

In caso di succeso 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

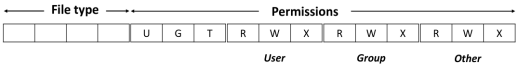
Si tratta di una bit mask, presente anche nella `struct stat` , che ci permette di definire i permessi dei file. Lunga 16 bit, i primi 9 sono per other, group e user, rispettivamente 3 a testa. Possiamo usarla per avere informazioni sulla tipologia del file. Esempio:

```
        char pathname[] = "/tmp/file.txt";
        struct stat statbuf;
        //Getting the attribute of /tmp/file.txt
        if(stat(pathname, &statbuf) == -1)
            errExit("stat");

        //Checking if /tmp/file.txt is a regular file
        if((statbuf.st_mode & S_IFMT) == S_IFREG)
            printf("regular file!\n);

        //Equivalently, checking if /tmp/file.txt is a
        //regular file by S_ISREG macro
        if(S_ISREG(statbuf.st_mode))
            printf("regular file!\n");

```



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 relativamento al nostro user id e group id.

```
        #include <unistd.h>

        //Returns 0 if all permissions are granted, otherwise
    -1
        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

```
        #include <unistd.h>

        //Returns 0 on success, or -1 on error.
        int rmdir(const char *pathname);

```

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

4.2.3 opendir, closedir e readdir

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

        //Returns directory stream handler, or NULL on error.
        DIR *opendir(const char *dirpath);

        //Returns 0 on success, or -1 on error
        int closedir(DIR *dirp);

```

Una volta aperta una directory per leggerla si userà

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

        //Returns pointer to an allocated structure
   describing the
        //next directory entry or NULL on end-of-directory or
    error
        struct dirent *readdir(DIR *dirp);

```

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-
    directory is reached!
14        if(errno != 0)
15        printf("Error while reading dir.\n");
16        closedir(dp);
17
```

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 <stlib.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 unseten(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
    error
4         int dup(int oldfd);
5
```

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`

```
#include <stdlib.h>
//Returns 0 on success, or nonzero on error

int atexit(void (*func)(void));

```

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

Esempio pratico

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

void func1(){
    printf("\tAtexit function 1 called\n");
}
void func2(){
    printf("\tAtexit function 2 called\n");
}

int main(int argc, char *argv[]){
    if(atexit(func1) != 0 || atexit(func2) != 0)
        _exit(EXIT_FAILURE);
    exit(EXIT:SUCCESS);
}

```

L'eventuala output sarà

```
Atexit function 2 called
Atexit function 1 called

```

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

```
#include <unistd.h>

//In parent: returns process ID of child on success, or -1 on error
//In created child: always returns 0
```

```
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 eslclusivamente 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)"
    : "(parent)", stack);
12        }
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 valoe di ritorno della `waitpid` è 0;

0 : aspetta solo per i figli che terminano.

```
        pid_t pid;
        for(int i = 0; i < 3; ++i){
            pid = fork();
            if(pid == 0){
                //Code executed by the child process...
                exit(0);
            }
        }
        //The parent process only waits for the last created
    child
        waitpid(pid, NULL, 0);

```

Altro esempio:

```
        pid_t pid = fork();
        if(pid == 0){
            //Code executed by the child process
        }
        else{
            //Waiting for a terminated/stopped | resumed
    child process
            waitpid(pid, NULL, WUNTRACED | WCONTINUED);
        }

```

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:

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

```

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

```

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.

```
        #include <unistd.h>
        //None of the following returns on success, all
return -1 on error
        int execl(const char *path, const char *arg, ...); //
variadic functions
        int execlp(const char *path, const char *arg, ...);
        int execle(const char *path, const char *arg, ...,
char *const envp[]);
        int execv(const char *path, char *const argv[]);
        int execvp(const char *path, char *const argv[]);
        int execve(const char *path, char *const argv[], char
 *const envp[]);

```

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

function	path	arguments (argv)	environment (envp)
exec**l**	pathname	list	caller's environ
exec**lp**	filename	list	caller's environ
exec**le**	pathname	list	array
exec**v**	pathname	array	caller's environ
exec**vp**	filename	array	caller's environ
exec**ve**	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",
    NULL};
8             execv("./hello", args);
9             printf("Back to example.c");
10
11            return 0;
12        }
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 chavi

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 sovrascrizione. 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(
   check errno)
4        key_t ftok(char *pathname, int proj_id);
5
```

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 struttua, 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 lavore 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 `strcut semid_ds`, verrà salvato all'interno di `arg.buf`;
- `IPC_SET`: occore 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         strcut 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:

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

```

Altre operazioni:

```
        int semctl(semid, semnum, cmd, 0);

```

- `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

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

        //Returns 0 on success, or -1 on error
        int semop(int semid, struct sembuf *sops, unsigned
    int nsops);

```

La struttura `sembuf` è la seguente:

```
        struct sembuf{
            unsigned shor sem_num; //semaphore number
            short sem_op; //operation to be performed
            short sem_flag; //operation flags
        }

```

`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 outpu `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	si	termina un processo
SIGINT	2	si	termina un processo
SIGQUIT	3	si	core dump + terminazione
SIGKILL	9	no	termina un processo
SIGSTOP	17	no	blocca un processo
SIGCONT	19	si	riprende un processo
SIGPIPE	13	si	termina un processo
SIGALRM	14	si	termina un processo
SIGUSR1	30	si	termina un processo
SIGUSR2	31	si	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 paramentro 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
    SIG_ERR on error
5         sighandler_t signal(int signum, sighandler_t handler)
    ;
6
```

`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
     SIGUSR1
7             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
    SIGINT/SIGUSR1
12            //is delivered, sigHandler will be used to handle
     the signal.
13            //Reset the default process disposition for
    SIGINT and SIGUSR1
14            if(signal(SIGINT, SIG_DFL) == SIG_ERR ||
15            signal(SIGUSR1, SIG_DFL) == SIG_ERR){
16                errExit("reset signal handler failed");
17            }
18            return 0;
19        }
20
```

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 proccesso 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 appartenti 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, sè 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
    process
11            }
```

```
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 risettare 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,
   sigset_t *oldset);
5
```

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.

6.6 Shared Memory

Segmento di memoria fisica condivisa gestita dal Kernel, una volta istanziata, viene linkata al processo che la vuole utilizzare. I dati scritti sulla memoria condivisa da un processo A sono immediatamente visti da un processo B. La mutua esclusione in questa casistica è importante per chi scrive (semafori).

6.6.1 shmget

```
1       #include <sys/shm.h>
2
3       //Returns a shared memory segment identifier on
    success, or -1 on error
4       int shmget(ket_t key, size_t size, int shmflg);
5
```

La `size` della shared memory è definita in byte e verrà sempre approssimato alla dimensione successiva della pagina di sistema. Se si prova a creare un segmento di memoria usando la chiave di un segmento già esistente, nel caso in cui si passa una dimensione inferiore, la `shmget` tornerà l'ID del segmento, nel caso in cui la dimensione sarà maggiore tornerà un errore. Nella `shmflg` si possono mettere tutte quelle relative alla `open` con una nota particolare nei confronti di:

- `IPC_CREAT`: se non esiste alcun segmento con quella chiave, allora crea un nuovo segmento;
- `IPC_EXCL`: insieme a `IPC_CREAT` fa fallire la `shmget` se esiste già un segmento con quella chiave specificata.

6.6.2 shmat

```
1       #include <sys/shm.h>
2
3       //Returns address at which shared memory is attached
    on success
4       //or (void *) -1 on error
5       int *shmat(int shmid, const void *shmaddr, int shmflg
    );
6
```

`shmaddr`, di norma è `NULL`, il Kernel deciderà quale indirizzo allocare, se invece non è NULL, si deve passare l'indirizzo a mano, in questo caso serve passare la `shmflg SHM_RND`, arrotonda l'indirizzo al multiplo più vicino alla costante `SHMLBA` (shared memory low boundary address). La flag serve ad arrotondare il più possibile nel caso in cui l'indirizzo passato sia occupato.

Perché `shmaddr` di base va messo a `NULL`?

- aumenta la portabilità dell'applicazione;
- cercare di mappare a mano rischia di mappare zone di memoria già occupate causando errori.

Tra le flag c'è anche la `SHM_RDONLY`, per poter leggere e basta sulla shared memory. Una volta che un processo ha fatto l'attach della shared memory, l'eventuale figlio creato successivamente avrà già a disposizione il puntatore a tale segmento. Eventualmente se venisse fatta una `exec` sul figlio, le shared memory verranno staccate perché non saranno più disponibili.

Esempio:

```
        //attach the shared memory in read/write mode
        int *ptr_1 = (int *)shmat(shmid, NULL, 0);
        //attach the shared memory in read only mode
        int *ptr_2 = (int *)shmat(shmid, NULL, SHM_RDONLY);
        //N.B. ptr_1 and ptr_2 are different!
        //But they refer to the same shared memory!
        //write 10 integers to shared memory segment
        for(int i = 0; i < 10; ++i)
            ptr_1[i] = i;
        //read 10 integers from shared memory segment
        for(int i = 0; i < 10; ++i)
            printf("integer: %d\n", ptr_2[i]);

```

6.6.3 shmdt

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

        //Returns 0 on success, or -1 on error
        int shmdt(const void *shmaddr);

```

Prende in input i puntatori al segmento ottenuti dopo la `shmat`. Dopo una detach la shared memory istanziata esiste ancora anche se non sarà più linkata, un ulteriore metodo è che tutti i processi che l'hanno richiamata terminino, una volta terminati la shared memory verrà eliminata. **N.B.:** la detach viene fatta in automatico alla terminazione del processo.

Esempio:

```
        //attach the shared memory in read/write mode
        int *ptr_1 = (int *)shmat(shmid, NULL, 0);
        if(ptr_1 == (void *) -1)
            errExit("first shmat failed");
        //attach the shared memory in read only mode
        int *ptr_2 = (int *)shmat(shmid, NULL, SHM_RDONLY);
        if(ptr_2 == (void *) -1)
            errExit("second shmat failed");
        //...
        //detach the shared memory segments
        if(shmdt(ptr_1) == -1 || shmdt(ptr_2) == -1)
```

```
12              errExit("shmdt failed");
13
```

6.6.4 shmctl

```
1       #include <sys/shm.h>
2
3       //Returns 0 on success, or -1 on error
4       int shmctl(int shmid, int cmd, struct shmid_ds *buf);
5
```

`cmd` sono:

- `IPC_RMID`: segna il segmento come da eliminare come tutti i processi attaccati si staccheranno;
- `IPC_STAT`: copia la struttura dati `shmid_ds` associata alla shared memory nella struttura puntata da `buf`;
- `IPC_SET`: update dei parametri della `shmid_ds` passati e associati alla shared memory tramite i valori passati da `buf`.

Per ogni shared memory il kernel ha una struttura dati a essa associata:

```
1       struct shmid_ds{
2           struct ipc_perm shm_perm; //Ownership and
    permissions
3           size_t shm_segsz; //Size of segment in bytes
4           time_t shm_atime; //Time of last shmat()
5           time_t shm_dtime; //Time of last shmdt()
6           time_t shm_ctime; //Time of last chane
7           pid_t shm_cpid; //PID of creator
8           pid_t shm_lpid; //PID of last shmat()/shmdt()
9           shmatt_t shm_nattch; //Number of currently
    attached processes
10      };
11
```

L'unico campo che si può modificare una volta inizializzata è il campo `shm_perm`.

6.7 Message Queue

Permette la comunicazione tra processi diversi.

6.7.1 msgget

```
1       #include <sys/msg.h>
2
3       //Returns message queue identifier on success, or -1
    on error
4       int msgget(key_t key, int msgflg);
5
```

Nel caso la coda fosse già stata creata, ritorna l'ID della coda già esistente. Per il parametro `msgflg` vale la stessa regola di quelle della shared memory.

Vengono inviate delle `struct`, un esempio è:

```
1       struct mymsg{
2           long mtype; //Message type
3           char mtext[]; //Message body
4       }
5
```

L'unico vincolo sulla creazione della `struct` è che sia presente il `long mtype`. Fondamentale perchè è un campo usato dalle system call della Message Queue per lettura e scrittura, il resto è scelto dal programmatore. **Deve** essere maggiore di 0.

6.7.2 msgsnd

```
1       #include <sys/msg.h>
2
3       //Returns 0 on success, or -1 on error
4       int msgsnd(int msqid, const void *msgp, size_t msgz,
    int msgflg);
5
```

Scrive il messaggio in una specifica cosa di messaggi, definita dall'ID. Necessario l'uso del puntatore (`*msgp`) al messaggio che si vuole inviare, dimensione (`msgz`) in byte ed eventuali flag. `msgflg` di norma è 0, così facendo la rende bloccante, ovvero se si prova a scrivere un messaggio su una coda piena, siccome la coda ha una dimensione massima, allora il processo si bloccherà, finchè un processo non leggerà un messaggio liberando spazio nella coda. Se invece non si vuole bloccare, la flag necessaria sarà `IPC_NOWAIT`, non si bloccherà e tornerà l'errore `EAGAIN`.

Esempio:

```
1       //Message structure
2       struct mymsg{
3           long mtype;
4           char mtext[100];
```

```
5         };
6         //Message has type 1
7         m.mtype = 1;
8         //Message contains the following string
9         char *text = "Ciao mondo!";
10        memcpy(m.mtext, text, strlen(text) + 1);
11        //size of m is only the size of its mtext attribute
12        size_t mSize = sizeof(struct mymsg) - sizeof(long);
13        //sending the message in the queue
14        if(msgsnd(msqid, &m, mSize, 0) == -1)
15            errExit("msgsnd failed");
16
```

N.B.: la coda di messaggi sa già di doversi aspettare un long, quindi va tolta la sua dimensione nell'invio.

6.7.3 msgrcv

```
1         #include <sys/msg.h>
2
3         //Returns number of bytes copied
4         //into msgp on success, or -1 on error
5         ssize_t msgrcv(int msqid, void *msgp, size_t msgz,
    long msgtype, int msgflg);
6
```

Legge il messaggio e allo stesso tempo **rimuove** il messaggio dalla coda liberando spazio. `*msgp` è il puntatore alla struttura in cui verrà salvato il messaggio. `msgsz` la dimensione del messaggio da leggere. `msgtype` permette di filtrare i messaggi alla ricerca di quelli con `mtype` specificato. Se `msgtype` è 0, si leggerà in ordine di arrivo, la lettura di default è FIFO. Infine se `msgtype` fosse negativo, verrà fatto l'assoluto del valore passato e si estrae dalla coda tutti i messaggi con valore minore o uguale a quello passato.

Se la coda è vuota o non c'è un messaggio con `mtype` specifico, il processo che fa la chiamata si blocca. Usando la flag `IPC_NOWAIT` nel campo `msgflg`, la lettura non sarà bloccante e si avrà in output l'errore `ENOMSG`, a indicare che non ci sono messaggi. Un'altra possibile flag è la `MSG_NOERROR`, di default se la dimensione del corpo del messaggio con `mtype` escluso, è più grande del `msgsz`, ritornerà un errore perché si proverà a leggere più byte del dovuto. Con questa flag, invece la `msgrcv` leggerà, ma darà comunque un errore in aggiunta.

6.7.4 msgctl

```
1         #include <sys/msg.h>
2
3         //Returns 0 on success, or -1 on error
4         int msgctl(int msqid, int cmd, struct msqid_ds *buf);
5
```

Permette di effettuare operazioni di controllo. Per il parametro `cmd` sono i soliti:

- `IPC_RMID`: rimuove immediatamente la coda di messaggi, tutti i messaggi non letti andranno perduti e ogni possibile processo in scrittura/lettura bloccato verrà sbloccato, settando `errno` con l'errore `EIDRM`. Per Questo `cmd` la `struct` viene ignorata;

- `IPC_STAT`: copia la `struct msqid_ds` associata alla coda di messaggi all'interno della `struct` puntata da `buf`;

- `IPC_SET`: aggiorna i campi della `struct msqid_ds` associata usando i valori passati tramite `buf`.

La `struct msqid_ds` è la seguente:

```
1         struct msqid_ds{
2             struct ipc_perm msg_perm; //Ownership and
    permissions
3             time_t msg_stime; //Time of last msgsnd()
4             time_t msg_rtime; //Time of last msgrcv()
5             time_t msg_ctime; //Time of last change
6             unsigned long __msg_cbytes; //Number of bytes in
    queue
7             msgqnum_t msg_qnum; //Number of message in queue
8             msglen_t msg_qbytes; //Maximum bytes in queue
9             pid_t msg_lspid; //PID of last msgsnd()
10            pid_t msg_lrpid; //PID of last msgrcv()
11        }
12
```

Con `IPC_STAT` e `IPC_SET` si può ottenere e modificare solo `msg_perm` e `msg_qbytes`.

6.8 Tabella riassuntiva

Interface	Message Queues	Semaphores	Shared memory
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/sem.h>`
Data structure	`msqid_ds`	`semid_ds`	`shmid_ds`
Create/Open	`msgget(...)`	`semget(...)`	`shmget(...)`
Close	none	none	`shdt(...)`
Control oper.	`msgctl(...)`	`semctl(...)`	`shmctl(...)`
Performing	`msgsnd(...)`	`semop(...)`	access memory
IPC	`msgrcv(...)`	to test/adjust	in shared region

6.9 PIPE

La PIPE è un byte stream, un buffer gestito dal kernel, viene creato al suo interno ed è gestito totalmente dal kernel, i processi lo utilizzano tramite il descriptor. Permette di inviare e leggere dati, è unidirezionale, la lettura avviene nell'ordine in cui si è scritto, diversamente dalle message queue. Non esiste un concetto di messaggio, PIPE e FIFO vengono trattate come dei semplici file. Come si crea una PIPE, il SO la istanzia, esiste ma è vuota, un processo che prova a leggere su una PIPE vuota si blocca, finchè un altro non ci scrive oppure qualche processo chiude la PIPE, in quel caso riceve un errore di `EINTR`.

Una PIPE ha due lati, uno di scrittura e uno di lettura, è fondamentale che **uno** solo legga e **uno** solo scriva. Non fare ciò può portare a errori. Se il lato di scrittura di una PIPE è chiuso o è stato chiuso, il processo in lettura vedrà una volta letto tutto un end-of-file, a segnalare che la PIPE è stata chiusa dal lato scrittura.

Un processo che cerca di scrivere sulla PIPE viene bloccato se non c'è più spazio disponibile sulla PIPE (la PIPE ha una dimensione fissa su Linux sono 65.536B), oppure riceve un segnale di terminazione.

La singola scrittura su una PIPE ha una dimensione `PIPE_BUF` massima di 4.096B, superare tale dimensione genera un errore. Sia su PIPE che su FIFO le scritture sono **atomiche**, non c'è possibilità che due messaggi si intersechino tra di loro.

6.9.1 pipe

L'apertura di un PIPE è equivalente alla `open` di un file, ma ovviamente con la sua system call dedicata.

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

```
3         //Returns 0 on success, or -1 on error
4         int pipe(int filedes[2]);
5
```

Prende in input un array vuoto, salverà all'interno di `filedes[0]` il lato di lettura, mentre in `filedes[1]` il lato di scrittura. Si tratta esattamente di un file descriptor che andrà a essere salvato all'interno della file descriptor table.

La lettura e scrittura verranno fatte tramite la `read` e la `write` per la gestione dei file, la prima su `filedes[0]` e la seconda su `filedes[1]`.

La **differenza principale** tra PIPE e FIFO è che la prima viene usata per far comunicare processi che hanno una relazione tra di loro. Per il semplice motivo che il programmatore non ha controllo sui file descriptor generati dalla `pipe`, perché sono gestiti dal SO, l'unico modo che un secondo processo sia a conoscenza di tali file descriptor è che sia figlio del primo processo e che quindi abbia ereditato tale informazioni.

Esempio:

```
1         int fd[2];
2         //checking if PIPE successed
3         if(pipe(fd) == -1)
4             errExit("PIPE");
5         //create a child process
6         switch(fork()){
7             case -1:
8                 errExit("fork");
9             case 0:
10                //child reads from PIPE
11                break;
12            default:
13                //parent writes to PIPE
14                break;
15        }
16
```

La chiusura di un lato della PIPE da parte di un processo e dell'altro da parte di un altro processo verrà fatta successivamente la `fork`, perché prima il figlio potrà ereditare il file descriptor completo dal padre e poi ognuno potrà modificarlo per sè.

Esempio di lettura da parte del figlio:

```
1         char buf[SIZE];
2         ssize_t nBys;
3
4         //close unused write-end
5         if(close(fd[1]) == -1)
6             errExit("close - child");
```

```
7         //reading from the PIPE
8         nBys = read(fd[0], buf, SIZE);
9         //0: end-of-file, -1: failure
10        if(nBys > 0){
11            buf[nBys] = '\0';
12            printf("%s\n", buf);
13        }
14        //close read-end of PIPE
15        if(close(fd[0]) == -1)
16            errExit("close - child");
17
```

Esempio di scrittura da parte del padre:

```
1         char buf[] = "Ciao Mondo\n";
2         ssize_t nBys;
3
4         //close unused read-end
5         if(close(fd[0]) == -1)
6             errExit("close - parent");
7         //writing to the PIPE
8         nBys = write(fd[1], buf, strlen(buf));
9         //checking if write successed
10        if(nBys != strlen(buf)){
11            errExit("write - parent");
12        }
13        //close write-end of PIPE
14        if(close(fd[1]) == -1)
15            errExit("close - parent");
16
```

N.B.: una PIPE continua a esistere fino a quando non vengono chiusi tutti i file descriptor in scrittura, perchè il kernel vede che è tuttora in uso e quindi non manda il carattere di terminazione.

6.10 FIFO

Sono byte stream anch'esse, dei buffer istanziati dal SO. La differenza principale tra FIFO e PIPE è che per la PIPE il nome viene generato in automatico dal SO e il programmatore non ha modo di far ottenere i file descriptor fra i diversi processi a meno che non siano padre e figlio o tramite altre IPC. Le FIFO ci permettono di dare un nome specifico alla creazione e poi due processi senza alcun legame tra di loro possono comunicare solo sapendo il nome della FIFO. Anche in questo caso, gli eventuali figli ereditano il file descriptor se fosse già stato creato dopo prima della `fork`.

6.10.1 mkfifo

```
#include <unistd.h>

//Returns 0 on success, or -1 on error
int mkfifo(const char *pathname, mode_t mode);

```

Il `pathname` è un path in cui noi decidiamo di andare a salvare la FIFO, il nome che si può scegliere per capirci. Le `mode` sono le stesse della system call `open`, permessi di scrittua, lettura, etc. Una volta creata la FIFO, la si apre tramite la `open`.

6.10.2 open

```
#include <unistd.h>

//Returns file descriptor on success, or -1 on error
int open(const char *pathname, int flags);

```

Si passa in input lo stesso `pathname` usato per la `mkfifo`. La FIFO può essere aperta in due modalità, lettura con `O_RDONLY` o scrittura `O_WRONLY`. Una volta ottenuto il file descriptor, si leggerà e scriverà tramite le `read` e la `write`.

Attenzione: dopo la creazione di una FIFO, se un processo A la apre, questo rimarrà bloccato finchè un processo B l'avrà aperta a sua volta. Ad ogni lettura o scrittura deve corrispondere un'apertura in scrittura o lettura. La FIFO potrebbe essere usata per sincronizzare i processi per esempio.

Esempio di lettura:

```
char *fname = "/tmp/myfifo";
int res = mkfifo(fname, S_IRUSR | S_IWUSR);
//Opening for reading only
int fd = open(fname, O_RDONLY);

//reading bytes from fifo
char buffer[LEN];
read(fd, buffer, LEN);

//printing buffer on stdout
printf("%s\n", buffer);

//closing the fifo
close(fd);

//removing fifo
```

```
17          unlink(fname);
18
```

Esempio di scrittura:

```
1           char *fname = "/tmp/myfifo";
2
3           //opening for writing only
4           int fd = open(fname, O_WRONLY);
5
6           //reading a str.(no spaces)
7           char buffer[LEN];
8           printf("Give me a string: ");
9           scanf("%s", buffer);
10
11          //writing the string on fifo
12          write(fd, buffer, strlen(buffer));
13
14          //closing the fifo
15          close(fd);
16
```

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 ritorno 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 basi 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:** Spiega 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.

6. **D:** Dove si trova la memoria condivisa? (shared memory)

R: Non si trova nel layout di memoria di un processo, ma nella memoria fisica del Kernel, poi il processo lo crea e richiede il link/attach a quella memoria. Se la memoria è condivisa in scrittura con altri processi occore garantire la mutua eslusione.

7. **D:** Come posso aumentare la dimensione della shared memory?

   R: Detacho ed elimino l'attuale e ne creo una nuova con una dimensione maggiore.

8. **D:** Cosa uso per far comunicare processo padre con processo figlio?

9. **D:** Se ho una processo A e un processo B che non hanno legami, posso usare la PIPE?