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 successo, 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
```

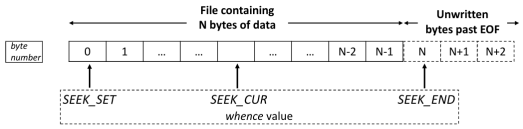
N.B.: `whence` indica la base di partenza dell'offset;

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

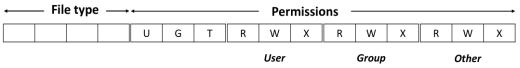
Si tratta di una bit mask, presente anche nella `struct stat` , che ci permette di definire i permessi dei file. Lunga 16 bit, i primi 9 sono per other, group e user, rispettivamente 3 a testa. Possiamo usarla per avere informazioni sulla tipologia del file. Esempio:

```
        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 descripto 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

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 mappa 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`