Indice

Capitolo 1 – Introduzione e fondamenti

In questo capitolo vengono introdotte le basi del laboratorio di Sistemi Operativi.

1.1 – Processi e programmi

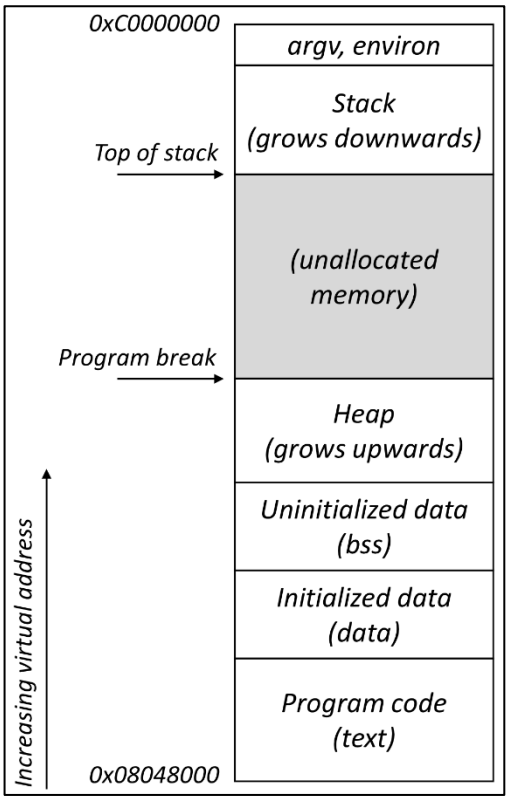
Un **processo** è un'istanza di un programma in esecuzione.

Un **programma** è un file binario contenente un insieme di informazioni che descrivono come costruire un processo ed inviarlo in esecuzione.

1.1.1 – Layout della memoria di un processo

Qui di seguito viene utilizzata la RAM di un processo nel sistema operativo Linux 32 *bit*.

- **Program code** è **utilizzabile solo in lettura** (*read-only*) ed è un segmento contenente le istruzioni macchina (*text*);
- **Initialized data** è il segmento contenente le variabili globali e/o statiche inizializzate (*data*);
- **Uninitialized data** è il segmento contenente le variabili globali e/o statiche **non** inizializzate (*bss*);
- **Heap** è il segmento contenente le variabili allocate dinamicamente, ovvero tramite la *malloc* per esempio. La particolarità di questo segmento è che si **espande verso** l'**alto** quando ha bisogno di memoria aggiuntiva andando verso "*unallocated memory*";
- **Stack** è il segmento contenente gli argomenti di ogni funzione invocata e delle variabili locali dichiarate. Analogamente allo *heap*, anche lo *stack* si **espande** quando necessita di memoria ma **verso** il **basso**.



Un **esempio** di codice per capire meglio.

```c
#include <stdlib.h>
// Declared global variables
char buffer[10];                    // <- (bss)
int primes [] = {2, 3, 5, 7};       // <- (data)
// Function implementation
void method (int *a)                // <- (stack)
{
    int i;                          // <- (stack)
    for (i = 0; i < 10; ++i)
        a[i] = i;
}
// Program entry point
int main (int argc, char *argv[]) // <- (stack)
{
    static int key = 123;           // <- (data)
    int *p;                         // <- (stack)
    p = malloc(10 * sizeof(int)); // <- (heap)
    method(p);
    free(p);
    return 0;
}
```

Nella dichiarazione delle variabili globali, la variabile *buffer* finirà nel *bss* (uninitialized data) perché non viene inizializzata ed è globale, mentre *primes* finirà nel *data* (initialized data) poiché inizializzata.
Successivamente, nella funzione *method* gli argomenti di quest'ultima finiranno nello *stack* insieme all'unica variabile dichiarata localmente: *i*.
Per concludere, nel *main* gli argomenti di quest'ultima e l'unico puntatore verranno salvati nello *stack*.
Invece, la variabile *key* viene salvata nel *data* poiché è statica e l'allocazione dinamica utilizzando la *malloc* viene salvata nello *heap*.

Si ricorda che tutto il codice viene tradotto in linguaggio macchina e inserito all'interno del segmento *text*.

Una **curiosità** da conoscere è il comando *size*, il quale permette di vedere la dimensione in *byte* di ciascun segmento. Questo è possibile farlo scrivendo la keyword *size* nel terminale seguito dall'eseguibile del codice.

Per esempio, il codice precedente produce un *size* del tipo:

```
user@localhost[~]$ size main
   text    data     bss     dec     hex   filename
   1695     628      24    2347     92b       main
```

1.1.2 – Introduzione alla File descriptor table

Per ogni processo generato, il *Kernel* mantiene una *File descriptor table*. Ogni voce della tabella è un *file descriptor*, ovvero un numero positivo che rappresenta una risorsa di *input* o *output* aperta dal processo.

Per convenzione, il sistema operativo crea di default tre *file descriptors* in **ogni nuovo processo**. Questi tre *file* sono:

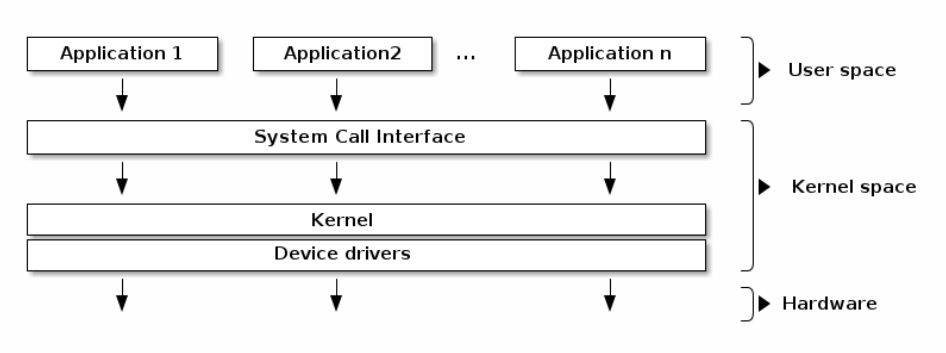
File descriptor	Purpose	POSIX name
0	*standard* **input**	STDIN_FILENO
1	*standard* **output**	STDOUT_FILENO
2	*standard* **error**	STDERR_FILENO

Per esempio, al momento di una creazione di un processo, egli avrà già i 3 *file descriptors* qua sopra citati. Inoltre, nel momento in cui verrà aperto un altro file da quel processo, esso verrà aggiunto alla tabella.

Tuttavia, se uno dei tre *file* di default viene chiuso, il prossimo *file* aperto prenderà il suo numero intero. Questo perché il numero assegnato dal *Kernel* deve essere il valore più piccolo dei presenti.

1.2 – System calls

Una *system call* è un punto d'entrata controllato (*controlled entry point*) situato all'interno del *Kernel*. La *system call* viene utilizzata dai processi per richiedere un servizio.



Per esempio, il servizio fornito dal *Kernel* include: la creazione di nuovi processi, l'esecuzione di operazioni di tipo *input/output*, la creazione di una "*pipe*" per la comunicazione tra processi, ecc.

Esiste anche un manuale che racchiude le *Linux system call*. Esso è reperibile direttamente da terminale scrivendo il comando man 2 per vedere le *syscall*.

```
MAN(1)                        Manual pager utils                        MAN(1)

NAME
       man - an interface to the system reference manuals

SYNOPSIS
       man [man options] [[section] page ...] ...
       man -k [apropos options] regexp ...
       man -K [man options] [section] term ...
       man -f [whatis options] page ...
       man -l [man options] file ...
       man -w|-W [man options] page ...

DESCRIPTION
       man  is  the system's manual pager.  Each page argument given to man is
       normally the name of a program, utility or function.  The  manual  page
       associated with each of these arguments is then found and displayed.  A
       section, if provided, will direct man to look only in that  section  of
       the  manual.   The  default action is to search in all of the available
       sections following a pre-defined order (see DEFAULTS), and to show only
       the first page found, even if page exists in several sections.

       The table below shows the section numbers of the manual followed by the
       types of pages they contain.

       1   Executable programs or shell commands
       2   System calls (functions provided by the kernel)
       3   Library calls (functions within program libraries)
       4   Special files (usually found in /dev)
       5   File formats and conventions, e.g. /etc/passwd
       6   Games
       7   Miscellaneous (including  macro  packages  and  conventions),  e.g.
           man(7), groff(7)
       8   System administration commands (usually only for root)
       9   Kernel routines [Non standard]
```

Link di approfondimento: The Linux Kernel.

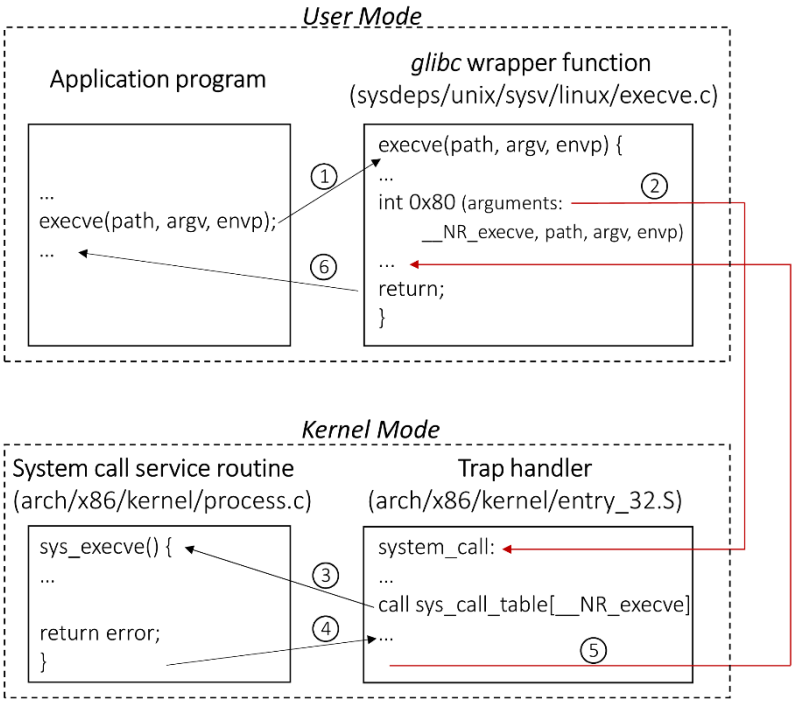
1.2.1 – Esecuzione delle System calls

Di seguito viene riportato un codice d'esempio per capire al meglio l'esecuzione di una *system call*.

Come **primissima** cosa viene eseguito il codice che contiene la *syscall*. La *system call* interessata è la *execve* la quale crea una nuova immagine di un processo ed elimina la precedente, quindi il contrario di una *fork*.

Al momento dell'esecuzione della *syscall*, l'osservazione passa al codice della *syscall* la quale contiene la keyword int 0x80, ovvero un *interrupt*!

Un *interrupt* permette di cambiare la modalità da utente (*User*) a *Kernel*. Questo cambio è essenziale per l'esecuzione della *service routine*.



Una volta eseguita la *routine* si ritorna indietro fino al codice iniziale ripristinando tutti i registri momentaneamente liberati a causa della *interrupt* (per approfondire si ripassi il corso di Architettura degli Elaboratori, capitolo sugli interrupt).

1.2.2 – Manipolazione degli errori delle system call

Guardando il codice del paragrafo precedente, si può osservare che viene ritornata una variabile chiamata *error*. Essa è presente in qualsiasi *syscall* e indica l'eventualità di un errore. Per un approfondimento è possibile consultare il manuale alla sezione "*ERRORS*".

Quando il valore di questa variabile è:

- $-1 \rightarrow$ si è verificato un **errore**
- *null pointer* $\rightarrow$ si è verificato un **errore**
- *valore positivo* $\rightarrow$ **non** si è verificato un **errore**

Per gestire gli errori all'interno del codice è opportuno importare la libreria $< errno.h >$ la quale permette di impostare una **variabile intera globale** chiamata ***errno***. Essa avrà un valore **positivo quando** verrà riscontrato un **errore** di una *syscall*.

Un **esempio** della gestione degli errori all'interno di un codice:

```c
#include <errno.h>
// ...
// system call to open a file
fd = open(pathname, flags, mode);
// BEGIN code handling errors.
if (fd == -1)
{
    if (errno == EACCES)
        // Handling not allowed access to the file
    else
        // Some other error occurred
}
// END code handling errors
// ...
```

Al momento della *syscall open* viene salvato il valore di ritorno, ovvero *error*, all'interno della variabile fd.

Se il valore è -1 significa che è stato riscontrato un errore. In caso di errore, viene effettuato un altro controllo, ma adesso sulla variabile *errno*. Il controllo verifica se è stata impostata al valore *EACCES*, ovvero se c'è stato un errore di permessi con il file indicato. In caso contrario, vuol dire che si è verificato un altro tipo di errore.

Tuttavia, talvolta è possibile che alcune *system call* restituiscano dei valori di successo nonostante la presenza di errori. Un **esempio** è la *syscall getpriority*:

```c
#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
}
// ...
```

Per controllare l'errore in questo caso, si inizializza la variabile *errno* e successivamente si controlla se è stato effettuato qualche cambiamento. Nel caso in cui sia stato effettuato un cambiamento, vuol dire che è stato riscontrato un errore.

Per **stampare** l'**errore** si utilizza la funzione *perror*( ), per esempio:

```c
#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);
}
// ...
```

Example output:

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

Quindi viene utilizzato il messaggio personalizzato per identificare l'errore e successivamente segue una piccola descrizione dell'errore incontrato.

Una **stampa più precisa** dell'**errore** è ottenuta grazie a *strerror* la quale accetta come argomento la variabile *errno*. Il valore ritornato dalla funzione è una stringa.

Attenzione! Se viene invocato più volte, la stringa può essere sovrascritta.

Un **esempio** di questa funzione:

```c
#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);
}
// ...
```

Example output:

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

In questo esempio la stampa dell'errore è molto più precisa.

Durante il laboratorio di Sistemi Operativi, i professori consigliano l'utilizzo della funzione creata da loro: *errExit*. Essa è una scorciatoia che permette di stampare un messaggio e di terminare il processo relativo.

La sua implementazione è la seguente:

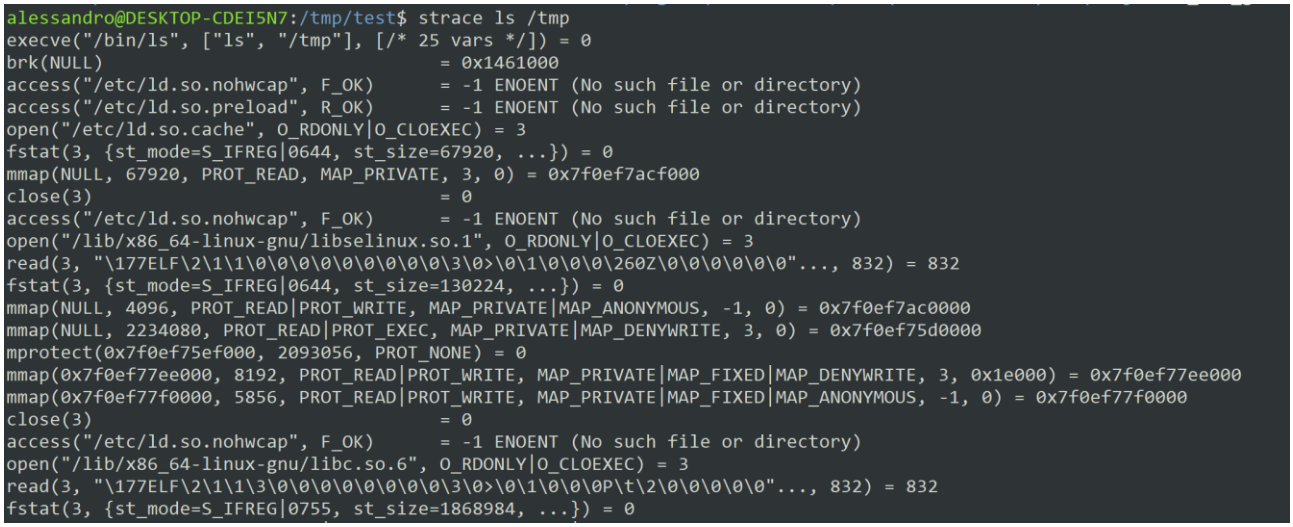
```c
void errExit (const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

Quindi, per utilizzarla è necessario implementarla scrivendola nel codice.

1.2.3 – Comando *strace*

La keyword *strace* rappresenta un comando di sistema che permette di scoprire quali *system call* sta utilizzando un processo in esecuzione.

Per utilizzarla basta aprire il terminale, inserire il comando *strace* e successivamente la *system call* relativa.



1.3 – Tipi di dato Kernel

A seconda della versione di Linux implementata, i tipi di dato utilizzati rappresentano informazioni differenti. Per esempio, nelle versioni Linux 2.2 e precedenti, gli utenti e gli IDs dei gruppi erano rappresentati in 16 $bits$, mentre nelle versioni 2.4 e successive, erano rappresentati in 32 $bits$.

Per risolvere tale problema, si utilizza la $typedef$ del linguaggio di programmazione C così da consentire uniformità tra le varie versioni.

La maggior parte dei tipi di dato standard del sistema hanno alla fine del nome i due caratteri "_t". Molti di questi tipi sono dichiarati nel $file\ header < sys/types.h >$. Per esempio, il tipo di dato pid_t è utilizzato per rappresentare gli IDs dei processi.

Nella seguente tabella vengono rappresentati alcuni tipi di dato più utilizzati durante il corso:

Data type	Type requirement	Description
ssize_t	signed integer	byte count or error indication
size_t	unsigned integer	byte count
off_t	signed integer	file offset
mode_t	integer	file permission and type
pid_t	signed integer	process, or process group, or session ID
uid_t	integer	numeric user identifier
gid_t	integer	numeric group identifier
key_t	arithmetic type	System V IPC type
time_t	integer or real floating	time in seconds since Epoch
msgqnum_t	unsigned integer	counts of messages in a queue
msglen_t	unsigned integer	number of allowed byte for a msg
shmatt_t	unsigned integer	counts attaches for a shared memory

1.4 – Pagine del manuale

Le pagine del manuale sono un insieme di pagine che descrivono ogni comando disponibile nel sistema indicando anche cosa fanno nello specifico. Per specifico si intende come utilizzare un comando, come avviarlo e quali argomenti immessi da linea di comando esso accetta.

Come detto in precedenza, il manuale è disponibile tramite il comando *man* seguito dal comando interessato.

Come si è visto per le *system call*, il manuale presenta diverse sezioni, qui di seguito le più importanti:

1. Comandi utente (esempio: man cd);
2. Documentazione delle *system call* (esempio: man 2 open);
3. Documentazione sulle funzioni di libreria fornite dalla libreria C standard (esempio: man 3 strlen);
4. Documentazione dettagliata dei dispositivi (esempio: man 4 hd);
5. Formati dei file e convenzioni (esempio: man 5 fstab).

1.5.1.1 – (*open*) Apertura/Creazione di un file

La *system call open* permette di aprire un file esistente o di crearlo e aprirlo. Come valore di ritorno si ha -1 solo in caso di errore, negli altri casi viene ritornato un *file descriptor* che viene utilizzato per riferirsi a quel determinato file nelle *system call* successive.

L'utilizzo generico:

```c
#include <sys/stat.h>
#include <fcntl.h>

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

Gli **argomenti** utilizzati sono:

- *pathname* → Necessario per aprire/creare un file poiché indica il percorso (*pathname*) all'interno del dispositivo;

- *flags* → Esso è un *bit mask*, ovvero un *bit maschera* che identifica quale o quali permessi di accesso si hanno a un determinato file. I flag sono molteplici, qui di seguito i più comuni:

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.

- *mode_t mode* → Sono i vari permessi che si concedono all'utente per maneggiare tale file. Come per il $flag$, anche *mode* è un *bit mask* con il quale si specificano i permessi da dare al nuovo $file$. I *mode* più comuni sono:

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

La **creazione di una maschera**, in gergo *user file-creation mask* (umask), è un attributo del processo che specifica quali *bits*, di permesso, devono essere impostati a zero alla creazione di un nuovo file da parte di un processo. Solitamente, la umask è impostata di default a 022 ovvero (– – – – w – –w –).

Nella pratica, i processi assegnati ad un nuovo file vengono paragonati tra la richiesta e la umask impostata. Per esempio:

```
Requested file perms:  rw-rw----  (<- this is what we asked)
Process umask:         ----w--w-  (<- this is what we are denied)
Actual file perms:     rw-r-----  (<- So, this is what we get)
```

La umask verrà approfondita in futuro.

Un **esempio** pratico che mostra l'utilizzo della *open*:

```
int fd;
// Open existing file for only writing.
fd = open("myfile", O_WRONLY);

// Open new or existing file for reading/writing, truncating
// to zero bytes; file permission read+write only for owner.
fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);

// Create and open a new file for reading/writing; file
// permissions read+write only for owner.
fd = open("myfile2", O_RDWR | O_CREAT | O_EXCL, S_IRUSR | S_IWUSR);
```

La **prima** *open* apre il file $myfile$ in sola scrittura.

La **seconda** *open* apre il file $myfile1$ in lettura e scrittura (O_RDWR), controlla se esiste e in caso di inesistenza lo crea (O_CREAT), azzera completamente il file (O_TRUNC), ovvero lo lascia vuoto. Invece, come permessi per l'utente è stato impostato il flag di lettura (S_IRUSR) e di scrittura (S_IWUSR).

La **terza** *open* apre il file $myfile2$ con gli stessi permessi e le stesse modalità utente del file $myfile1$ con l'unica eccezione del flag O_EXCL, ovvero il sistema si assicura che il file venga creato nel caso in cui non esista. In altre parole, controlla la corretta esecuzione di O_CREAT.

1.5.1.2 – (*read*) Lettura da un file descriptor

La *system call read* permette di leggere da un file descriptor. Il **valore di ritorno** è il numero di caratteri ancora presenti all'interno del file. Quindi, se il valore ritornato è -1 significa che si è presentato un errore, se è 0 significa che è stata raggiunta la fine del file (*end-of-file*, EOF), altrimenti la lettura è andata a buon fine.
L'utilizzo generico:

```c
#include <unistd.h>

// Returns number of bytes read, or -1 on error
ssize_t read(int fd, void *buf, size_t count);
```

Gli **argomenti** sono:

- fd → Rappresenta il *file descriptor* da cui leggere, per esempio STDIN_FILENO o STDOUT_FILENO (pagina 5);
- buf → È un puntatore ad un registro *buffer* utilizzato per salvare il risultato della lettura, precisamente è un puntatore ad un indirizzo di memoria nel quale viene salvato momentaneamente il risultato;
- $count$ → Indica il numero massimo di *bytes* da leggere dal file.

Esempio di una **lettura** di un **file** fino alla dimensione MAX_READ:

```c
// 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");
```

All'inizio c'è l'apertura di un file e il controllo di eventuali errori. Successivamente viene creato un buffer che ha una dimensione pari a MAX_READ + 1. Viene sommato l'uno poiché deve essere considerato anche il carattere terminatore, quindi posto alla fine, di qualsiasi stringa, ovvero "\0".

Infine, viene finalmente letto con la *syscall read* il file specificando il numero di *byte* con la variabile MAX_READ, con fd il file da leggere e con $buffer$ la destinazione degli elementi letti.

Un altro **esempio** è la **lettura** di caratteri da **terminale** e non da file:

```c
// 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);
```

La differenza dal codice precedente è la mancanza della *syscall open* poiché i caratteri interessati si trovano nel terminale e non in un file specifico. Quindi, si utilizza la *read* specificando come file il file descriptor STDIN_FILENO creato di default ogni volta che viene creato un processo.

1.5.1.3 – (*write*) Scrittura in un file descriptor

La *system call write* permette di scrivere informazioni in un file descriptor e come valore **ritorna** il numero di caratteri scritti.

L'utilizzo generico:

```c
#include <unistd.h>

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

Gli **argomenti** sono:

- *fd* → Rappresenta il *file descriptor* da cui leggere, per esempio STDIN_FILENO o STDOUT_FILENO (pagina 5);
- *buf* → Puntatore ad un registro *buffer* in cui ci sono i dati da scrivere nel file;
- *count* → Rappresenta il numero di *byte* del registro **puntato** da *buf*.

Esempio di **scrittura** della stringa "Ciao Mondo" in un **file**:

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

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

// Writing up to sizeof(buffer) bytes into myfile.
ssize_t numWrite = write(fd, buffer, sizeof(buffer));
if (numWrite != sizeof(buffer))
    errExit("write");
```

Viene aperto il file in modalità scrittura, viene creato un *buffer* con i dati da inserire all'interno del file e infine vengono scritti i caratteri all'interno del file. Si noti l'utilizzo della keyword *sizeof* per specificare la grandezza in *byte* del registro contenente i dati.

Mentre, un **esempio** di una **scrittura** di una stringa nel **terminale**:

```
// A buffer collecting a string.
char buffer[] = "Ciao Mondo";

// Writing up to sizeof(buffer) bytes on STDOUT.
ssize_t numWrite = write(STDOUT_FILENO, buffer, sizeof(buffer));
if (numWrite != sizeof(buffer))
    errExit("write");
```

1.5.1.4 – (*lseek*) Cursore all'interno del file

Quando si opera all'interno di un file, un cursore si muove all'interno di esso. Difatti, per ogni file aperto, il *kernel* salva un $file\ offset$, ovvero la posizione esatta in cui la prossima *read* o *write* inizierà ad operare. La *system call lseek* indica il cursore all'interno del file aperto e può essere modificato.

L'utilizzo generico:

```c
#include <unistd.h>

// Returns the resulting offset location, or -1 on error.
off_t lseek(int fd, off_t offset, int whence);
```

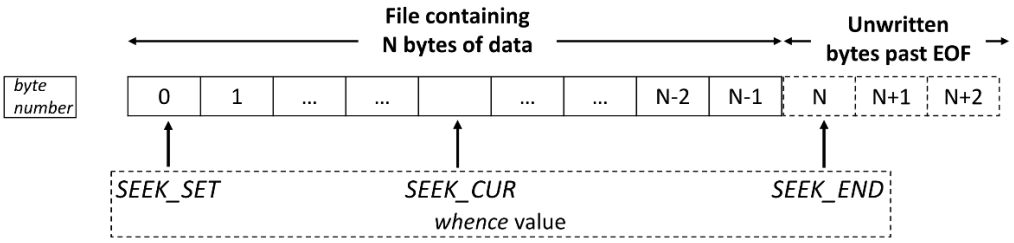
Gli **argomenti** sono:

- fd → Indica il $file\ descriptor$ del file aperto;
- $offset$ → Indica un valore in byte;
- *whence* → Indica il punto dal quale $offset$ è da interpretare.

Per comprendere, si lascia un **esempio**:

```c
// first byte of the file.
off_t current = lseek(fd1, 0, SEEK_SET);
// last byte of the file.
off_t current = lseek(fd2, -1, SEEK_END);
// 10th byte past the current offset location of the file.
off_t current = lseek(fd3, -10, SEEK_CUR);
// 10th byte after the current offset location of the file.
off_t current = lseek(fd4, 10, SEEK_CUR);
```

E il suo relativo schema:



La **prima** *syscall* indica il cursore all'inizio del file. Questo è caratterizzato dallo 0 come $offset$ e da SEEK_SET come $whance$.

La **seconda** *syscall* indica il cursore alla fine del file. Questo è caratterizzato dal -1 come $offset$ e da SEEK_END come $whance$.

La **terza** *syscall* indica 10 $byte$ precedenti rispetto alla posizione attuale del cursore. Questo è caratterizzato dal -10 come $offset$ e da SEEK_CUR come $whance$.

La **quarta** è come la terza ma con l'unica differenza che il $+10$ nel $offset$ indica 10 $byte$ successivi alla posizione attuale del cursore.

1.5.1.5 – (*close*) Chiusura di un file descriptor

La *system call close* permette di chiudere un file descriptor. Nonostante la chiusura non sia obbligatoria poiché qualsiasi *file descriptor* viene chiuso automaticamente nel momento in cui un processo termina, la *sycall close* è una "*good practice*". Questa pratica rende il codice comprensibile, leggibile e più ottimizzato per eventuali implementazioni future.

L'utilizzo generico:

```c
#include <unistd.h>

// Returns 0 on success, or -1 on error.
int close(int fd);
```

L'unico **argomento** presente è *fd* che rappresenta il *file descriptor* da chiudere.

1.5.1.6 – (*unlink*) Rimozione di un file

La *system call unlink* permette di rimuovere un *link* (*link* creato quando un processo ha il file nella sua *file descriptor table*) dal file e nel caso in cui esso sia l'ultimo *link* del file, allora verrà rimosso anche il file stesso. Attenzione perché questa *syscall* **non** rimuove una cartella, per questo si utilizza *rmdir*.

L'utilizzo generico:

```c
#include <unistd.h>

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

L'unico **argomento** è *pathname*, un puntatore al percorso del file da rimuovere.

Un **esempio** rapido con la creazione di un file, la chiusura e la rimozione:

```c
// Create a new file named myFile.
int fd = open("myFile", O_CREAT | O_WRONLY);
// ... only writes as myFile is open in write-only
// Close the file descriptor fd
close(fd);
// Unlink (remove) myFile
unlink("myFile");
```

1.5.2.1 – (*stat*, *lstat*, *fstat*) Recuperare gli attributi di un file

Le tre *system call stat*, *lstatm fstat* permettono di recuperare alcune informazioni da un file. Precisamente:

- *stat* → Ritorna informazioni riguardo il nome del file;
- *lstat* → Ritorna informazioni riguardo al link simbolico, ovvero un collegamento;
- *fstat* → Ritorna informazioni riguardo al *file descriptor* collegato a quel file.

L'utilizzo generico:

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

// Return 0 on success or -1 on error.
int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

Le **informazioni** ritornate sono sotto forma di **struttura**. Essa ha una composizione di questo tipo:

```
struct stat
{
    dev_t st_dev;         // IDs of device on which file resides.
    ino_t st_ino;         // I-node number of file.
    mode_t st_mode;       // File type and permissions.
    nlink_t st_nlink;     // Number of (hard) link to file.
    uid_t st_uid;         // User ID of file owner.
    gid_t st_gid;         // Group ID of file owner.
    dev_t st_rdev;        // IDs for device special files.
    off_t st_size;        // Total file size (bytes).
    blksize_t st_blksize; // Optimal block size for I/O (bytes).
    blkcnt_t st_blocks;   // Number of (512B) blocks allocated.
    time_t st_atime;      // Time of last file access.
    time_t st_mtime;      // Time of last file modification.
    time_t st_ctime;      // Time of last status change.
};
```

I campi hanno questo significato:

- **IDs dei dispositivi e numero *i-node***:
  - *st_dev* → identifica il dispositivo nel quale il file risiede
  - *st_ino* → contiene il numero *i-node* del file

  La combinazione di questi due campi identifica un file attraverso tutti i *file system*.

- **Proprietà del file**:
  - *st_uid* → identifica lo *user ID* al quale il file appartiene
  - *st_gid* → identifica il *group ID* al quale il file appartiene

- **Contatore di link**:
  - *st_nlink* → è il numero di *links* creati sul file

- **Timestamps dei file**:
  - *st_atime* → contiene il tempo dall'ultimo accesso al file
  - *st_mtime* → contiene il tempo dall'ultima modifica al file
  - *st_ctime* → contiene l'ultimo cambiamento al file

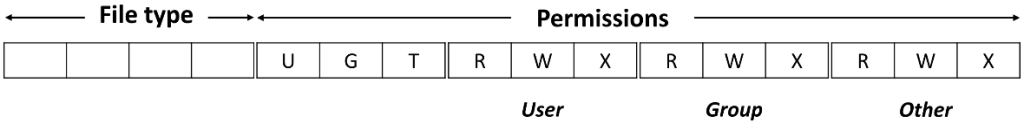
  I tempi sono espressi in secondi.

- **Dimensione del file**, **allocazione dei blocchi e dimensione ottimali dei blocchi di I/O**:
  - *st_size* → contiene la dimensione totale del file in *byte* e nel caso di un link simbolico, contiene la lunghezza in *byte* del percorso puntato dal link
  - *st_blocks* → contiene il numero di blocchi attualmente allocati per il file
  - *st_blksize* → contiene la dimensione, in *byte*, del blocco ottimale per I/O nei file del *file system* rispettivo. Un valore tipico è 4096.

1.5.2.2 – Tipi di file e permessi

Nella precedente struttura era presente il campo *st_mode*. Esso è un *bit mask* con il duplice obbiettivo di identificare il tipo di file e di specificare i permessi da concedere.

La struttura del campo *st_mode* è formata come segue:



Il **tipo di file** viene indicato nei primi 4 *bits* e può essere calcolato in due modi:

1. Eseguendo un'operazione logica di tipo *and* con la costante S_IFMT e successivamente, una comparazione tra il risultato ottenuto e un intervallo di costanti. Le **costanti** con cui eseguire la comparazione sono:

Constant	Test macro	File type
S_IFREG	S_ISREG( )	Regular file
S_IFDIR	S_ISDIR( )	Directory
S_IFCHR	S_ISCHR( )	Character device
S_IFBLK	S_ISBLK( )	Block device
S_IFIFO	S_ISIFO( )	FIFO or pipe
S_IFSOCK	S_ISSOCK( )	Socket
S_IFLNK	S_ISLNK( )	Symbolic link

   Quindi, per esempio, se dopo l'operazione logica *and* il risultato corrisponde a S_IFREG, vuol dire che il file è di tipo *regular*, ovvero un file *txt*.

2. Invocando direttamente una **macro** e passandogli come argomento il campo *st_mode*. Se il valore di ritorno è 0, allora il file sarà di quel tipo, altrimenti no.

Un **esempio** con entrambi i metodi di calcolo.

```c
char pathname[] = "/tmp/file.txt";
struct stat statbuf;
// Getting the attributes 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");
```

Prima di tutto c'è la dichiarazione di un file, successivamente la dichiarazione della struttura di tipo *stat* e infine il controllo per verificare eventuali errori.

Il controllo degli errori viene effettuato tramite la *syscall stat* che prende in input il file e il buffer.

Gli ultimi due costrutti condizionali sono le due operazioni per controllare il tipo di file. La **prima** utilizza l'operazione *and* logico per controllare *st_mode*

con la costante S_IFMT e successivamente controlla se il risultato è uguale alla costante S_IFREG, ovvero se è uguale a un file $regular$. Al contrario, la **seconda** utilizza direttamente la macro S_ISREG passandogli come argomento la st_mode. Il risultato non cambia.

I restanti 12 bit di permesso sono divisi in più gruppi. I **primi 3 *bits*** sono:

- **I *bits* U e G sono applicati con file eseguibili** (esempio . exe):
  - $set\text{-}user\text{-}ID$: verifica se l'effettivo $user\ ID$ del processo è lo stesso del proprietario dell'eseguibile; quindi, se l'utente è il proprietario dell'eseguibile;
  - $set\text{-}group\text{-}ID$: verifica se l'effettivo $group\ ID$ del processo è lo stesso del proprietario dell'eseguibile.
- **Il *bit* T chiamato *Sticky-bit***:
  - Utilizzato per evitare che una cartella possa essere eliminata o rinominata ($restricted\ deletion$). Non verrà utilizzata nel corso, dunque non è necessario approfondirla.

Mentre i **restanti 9 *bits*** definiscono i permessi di accesso ai file dalle varie categorie di utenti. Esistono 3 gruppi:

- ***User*** → I permessi concessi al proprietario del file;
- ***Group*** → I permessi concessi agli utenti che sono membri di quel gruppo di file;
- ***Other*** → I permessi concessi a tutti gli altri.

Ogni gruppo contiene **3 permessi**:

1. ***Read*** → Il contenuto del file può essere letto;
2. ***Write*** → Il contenuto del file può essere cambiato;
3. ***Execute*** → Il file può essere eseguito.

Le **cartelle** hanno gli stessi permessi dei file. Quindi in *lettura* vuol dire che è possibile vedere la lista del suo contenuto, in *scrittura* vuol dire che è possibile creare ed eliminare file al suo interno, in *esecuzione* vuol dire che è possibile accedere ai file dentro la directory.

Attenzione! Quando si vuole eseguire un'operazione di scrittura/lettura/esecuzione su un file, il permesso di accesso è richiesto da tutte le carelle che formano il percorso intero (in gergo $pathname$). Per esempio, se si vuole leggere il file con $pathname$:

$$/home/user1/secrets/passwords.txt$$

Allora si dovrà avere il permesso di esecuzione per cartelle:

- $/$
- $/home$
- $/user1$
- $/secrets$

Per **verificare i permessi** di un file tramite codice è possibile implementare il file *header* chiamato $< sys/stat.h >$ che definisce alcune costanti:

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Il loro utilizzo è simile a quello del tipo di file; quindi, si esegue un'operazione logica di tipo *and* tra *st_mode* e una delle costanti fornite dal file *header*.

Un **esempio** chiarificatore:

```
char pathname[] = "/tmp/file.txt";
struct stat statbuf;

// Getting the attributes for the executable /tmp/a.out
if (stat(pathname, &statbuf) == -1)
    errExit("stat");

// printing out the user's permissions
printf("user's permissions: %c%c%c\n",
    (statbuf.st_mode & S_IRUSR)? 'r' : '-',
    (statbuf.st_mode & S_IWUSR)? 'w' : '-',
    (statbuf.st_mode & S_IXUSR)? 'x' : '-');
```

In questo caso viene utilizzato il costrutto condizionale in una sola riga per verificare ciascuna condizione.

1.5.2.3 – (*access*) Controllare l'accessibilità di un file

La *system call access* permette di controllare l'accessibilità di un file.

L'utilizzo generico:

```c
#include <unistd.h>

// Returns 0 if all permissions are granted, otherwise -1
int access(const char *pathname, int mode);
```

Gli **argomenti** sono due:

- *pathname* → Si riferisce al percorso del file;
- *mode* → È un *bit mask* e consiste in una o più delle seguenti costanti:

Flag	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?

Un **esempio**:

```c
char pathname[] = "/tmp/file.txt";

// Checking if /tmp/file.txt exists, can be read and
// written by the current process.
if (access(pathname, F_OK | R_OK | W_OK) == -1)
    printf(" It looks like that I cannot read/write file.txt :(\n)");
```

1.5.2.4 – (*chmod*) Cambiare i permessi di un file

La *system call chmod* e *fchmod* permettono di cambiare i permessi di un file.

L'utilizzo generico di entrambe:

```
// All return 0 on success, or -1 on error
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);

#define _BSD_SOURCE
#include <sys/stat.h>

int fchmod(int fd, mode_t mode);
```

Gli **argomenti** sono due: *pathname* e *mode*. Nella *fchmod* bisogna inserire il *file descriptor* invece che il *pathname*. Gli argomenti non vengono approfonditi poiché identici ad *access*.

Un **esempio** comune:

```
char pathname[] = "/tmp/file.txt";

struct stat sb;
if (stat(pathname, &sb) == -1)
    errExit("stat");

// Owner-write on, other-read off, remaining bits unchanged unchanged.
mode_t mode = (sb.st_mode | S_IWUSR) & ~S_IROTH;

if (chmod(pathname, mode) == -1)
    errExit("chmod");
```

Nel momento in cui si inizializza *mode*, si esegue l'operazione di *or* logico tra la *st_mode* e S_IWUSR, ovvero la possibilità di scrivere nel file da parte dell'utente (si guardi la tabella nel paragrafo dei tipi di file e i permessi), e infine l'operazione di *and* logico con S_IROTH per indicare di lasciare tutti gli altri bit di sistema invariati.

Infine, il costrutto condizionale per verificare il valore di ritorno.

1.6.1.1 – (*mkdir*) Creazione di una nuova cartella

La *system call mkdir* viene utilizzata per creare una nuova cartella.

L'utilizzo generico:

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

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

I due **argomenti**:

- *pathname* → Specificare la collocazione della nuova cartella tramite *pathname*, quindi inserendo il percorso relativo o assoluto;
- *mode* → Specificare i permessi della nuova cartella creata.

Nel caso in cui la cartella esistesse già, la *syscall mkdir* torna l'errore EEXIST.

1.6.1.2 – (*rmdir*) Eliminazione di una cartella

La *system call rmdir* viene utilizzata per eliminare una cartella.

L'utilizzo generico:

```c
#include <unistd.h>

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

L'unico **argomento** da inserire è ovviamente il percorso della cartella.

Per il corretto funzionamento, la cartella per essere eliminata **deve essere vuota**.

Se l'ultimo componente del *pathname* è un *symbolic link*, esso non viene deferenziato, quindi ci sarà l'errore ENOTDIR.

Un **esempio** rapido:

```
// Create a new directory with name myDir.
int res = mkdir("myDir", S_IRUSR | S_IXUSR);
if (res == 0)
{
    printf("The directory myDir was created!\n");

    // Remove the directory with name myDir
    res = rmdir("myDir");
    if (res == 0)
        printf("The directory myDir was removed!\n");
}
```

1.6.1.3 – (*opendir*, *closedir*) Apertura e chiusura di una cartella

La *system call opendir* e *closedir* vengono utilizzate per aprire e chiudere una cartella.

L'utilizzo generico:

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

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

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

A differenza delle *syscall* viste finora, la *opendir* ritorna un puntatore al primo elemento della lista degli elementi all'interno della cartella. Quindi supponendo che esistano 3 file all'interno della cartella, con *opendir* è possibile aprire la cartella e puntare al primo elemento.

La *closedir* chiude semplicemente la risorsa di apertura ammettendo come parametro il puntatore utilizzato all'apertura.

Un **esempio** sarà effettuato nel paragrafo successivo.

1.6.1.4 – (*readdir*) Lettura di una cartella

La *system call readdir* viene utilizzata per leggere il contenuto all'interno di una cartella.

L'utilizzo generico:

```c
#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);
```

Come **argomento** necessita solo del puntatore al primo elemento della cartella aperta.

La struttura ritornata viene chiamata *dirent* e ha la seguente forma:

```c
struct dirent
{
    ino_t d_ino;          // File i-node number.
    unsigned char d_type; // Type of file.
    char d_name[256];     // Null-terminated name of file.
    //...
};
```

Il valore *d_type* può avere diversi significati, ma per fortuna in C sono state definite alcune macro che aiutano la comprensione:

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

Un **esempio** con l'apertura, la lettura e la chiusura:

```c
DIR *dp = opendir("myDir");
if (dp == NULL)
    return -1;

errno = 0;
struct dirent *dentry;
// Iterate until NULL is returned as a result.
while ((dentry = readdir(dp)) != NULL)
{
    if (dentry->d_type == DT_REG)
        printf("Regular file: %s\n", dentry->d_name);
    errno = 0;
}
// NULL is returned on error, and when the end-of-directory is reached!
if (errno != 0)
    printf("Error while reading dir.\n");
closedir(dp);
```

Prima di tutto viene aperto il file *myDir*. In caso il puntatore fosse NULL vorrebbe dire che non esiste.

Successivamente vengono letti gli elementi all'interno della cartella. La lettura prosegue finché la *syscall readdir* non torna il valore NULL. Essendo più specifici, vengono letti **solamente** i *regular file* poiché è stato specificato nel costrutto condizionale *if* tramite la macro DT_REG.

Infine, la cartella viene chiusa grazie a *closedir*.

Capitolo 2 – Manipolazione di un processo

In questo capitolo vengono introdotte alcune *system call*.

1.1 – (*getpid*) Identificatore di un processo

La *system call getpid* ritorna il valore ID di un processo specifico.

L'utilizzo generico:

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

pid_t getpid(void);
```

Il tipo di dato utilizzato per dichiarare il valore ritornato, ovvero *pid_t*, è un intero dovuto al processo di memorizzazione degli ID.

Attenzione! Non esiste alcuna relazione tra un programma e lo *ID process* del processo che ha messo in esecuzione il programma stesso. Esistono poche eccezioni come *init* che ha il *process ID* pari a 1.

N.**B**. La *syscall getpid* **non** ha bisogno di controlli poiché ritorna sempre il valore richiesto.

1.2 – (*getuid*, *geteuid*/*getgid*, *getegid*) ID *user* e ID *group* reale ed effettivo di un processo

Le *system calls getuid* e *getgid* ritornano rispettivamente il valore *ID user* reale e il valore *ID group* reale del processo chiamante.

Invece, le *system calls geteuid* e *getegid* ritornano rispettivamente il valore *ID user* effettivo e il valore *ID group* effettivo del processo chiamante.

L'utilizzo generico:

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

uid_t getuid(void);  // Real user ID
uid_t geteuid(void); // Effective user ID

gid_t getgid(void);  // Real group ID
gid_t getegid(void); // Effective group ID
```

La **distinzione** tra *reale* ed *effettiva*:

- **ID reale**, identifica l'utente e il gruppo al quale il processo appartiene;
- **ID effettivo**, utilizzato per determinare i permessi garantiti ad un processo quando esso prova ad effettuare alcune operazioni;

Per **esempio**, con il seguente codice:

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

int main (int argc, char *argv[])
{
    printf("PID: %d, user-ID: real %d, effective %d\n", getpid(), getuid(),
geteuid());
    return 0;
}
```

Si ha il seguente risultato:

```
user@localhost[~]$ gcc -o program program.c
user@localhost[~]$ ls -l program
   -r-xr-xr-x 1 Professor Professor 8712 Jan 16 16:27 program

user@localhost[~]$ ./program
   PID: 1234, user-ID: real 1000, effective 1000

user@localhost[~]$ sudo ./program
   PID: 1423, user-ID: real 0, effective 0

user@localhost[~]$ sudo chmod u+s program
user@localhost[~]$ ls -l program
   -r-sr-xr-x 1 root Professor 8712 Jan 16 16:27 program

user@localhost[~]$ ./program
   PID: 4321, user-ID: real 1000, effective 0
```

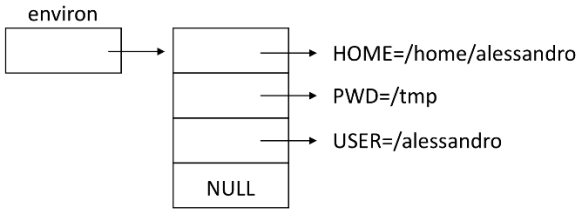
1.3 – (*getenv, setenv, unsetenv*) Ambiente di un processo

Ogni processo ha associato un *array* di stringhe chiamato lista d'ambiente o *environment list*. Ogni stringa è formata nel seguente modo:

$$name = value$$

Quando un nuovo processo viene creato, esso eredita una copia della lista d'ambiente del padre.

Un **esempio** di *environment list*:



L'**accesso** a questa lista è possibile in due modi:

1. Utilizzando la variabile globale *char **environ*.

Per esempio:

```c
#include <stdio.h>
// Global variable pointing to the enviroment of the process.
extern char **environ;

int main (int argc, char *argv[])
{
    for (char **it = environ; (*it) != NULL; ++it)
        printf("--> %s\n", *it);

    return 0;
}
```

```
user@localhost[~]$ ./program
--> $HOME=/home/Professor
--> $PWD=/tmp
--> $USER=Professor
```

2. Ricevere la lista d'ambiente come terzo argomento della funzione *main*. Attenzione che questa tecnica non è riconosciuta da tutti i sistemi operativi
Per esempio:

```c
#include <stdio.h>
int main (int argc, char *argv[], char* env[])
{
    for (char **it = env; (*it) != NULL; ++it)
        printf("--> %s\n", *it);

    return 0;
}
```

```
user@localhost[~]$ ./program
--> $HOME=/home/Professor
--> $PWD=/tmp
--> $USER=Professor
```

Dopo aver introdotto la definizione di *environment list*, adesso si osserveranno le 3 *system call* utilizzate per modificare la lista d'ambiente.

La *syscall getenv* accetta come parametro il nome di una variabile e ritorna un puntatore al valore della stringa di un *environment* con tale nome. Se non dovesse esistere, il valore ritornato sarebbe NULL.

La *syscall setenv* aggiunge "*name* = *value*" al *environment*, ovvero crea o modifica un *environment*. Nel caso di modifica e/o creazione, come parametri accetta il nome della variabile del *environment* esistente, il valore da applicare e l'eventuale valore di *overwrite* posto a 1 per sovrascrivere i valori già esistenti.

La *syscall unsetenv* rimuove la variabile identificata con il nome specifico dalla lista delle variabili d'ambiente (*environment list*).

L'utilizzo generico:

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

1.4 – (*getcwd, chdir, fchdir*) Cartella di lavoro (*working directory*)

Con la *system call getcwd* è possibile ottenere la posizione (*directory*) all'interno del sistema operativo, del processo in esecuzione.

L'utilizzo generico:

```c
#include <unistd.h>

// Returns cwdbuf on success, or NULL on error.
char *getcwd(char *cwdbuf, size_t size);
```

Come **argomenti** ha un buffer in cui scrivere, come stringa, il percorso della cartella in cui è contenuto il processo e la dimensione di tale buffer.

Il valore **ritornato** è un puntatore al registro, ovvero *cwdbuf*, in cui vi è scritto il percorso della cartella in cui è contenuto il processo. Nel caso in cui ci fosse un errore, per esempio la *working directory* supera la dimensione del buffer, la *syscall* ritorna NULL.

Con la *system call chdir* è possibile cambiare la cartella di lavoro in cui il processo sta lavorando.

L'utilizzo generico:

```c
#include <unistd.h>

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

Come **argomento** deve essere specificato il nuovo percorso che si intende inserire. Il percorso deve essere inserito per esteso!

Analogamente, la *system call fchdir* semplifica la possibilità di modifica della cartella, tramite i files descriptor. Difatti, come **argomento** essa richiede un file descriptor ottenuto precedentemente tramite una *syscall* di tipo *open* sulla cartella interessata.

Il suo utilizzo generico:

```c
#define _BSD_SOURCE
#include <unistd.h>

// Returns 0 on success, or -1 on error.
int fchdir(int fd);
```

Un **esempio** contenente queste tre *system call*:

```c
char buf[PATH_MAX];
// Open the current working directory
int fd = open(".", O_RDONLY);
getcwd(buf, PATH_MAX);
printf("1) Current dir:\n\t%s\n", buf);

// Move the process into /tmp
chdir("/tmp");
getcwd(buf, PATH_MAX);
printf("2) Current dir:\n\t%s\n", buf);

// Move the process back into the initial directory
fchdir(fd);
getcwd(buf, PATH_MAX);
printf("3) Current dir:\n\t%s\n", buf);

// Close the file descriptor
close(fd);
```

1.5 – (*dup*) Manipolazione della file descriptor table

Ogni processo ha un file descriptor table associato. La cartella */proc/*<PID>*/fd* contiene un link simbolico per ogni file descriptor presente nella tabella. Ovviamente, al posto di PID deve essere inserito l'ID preciso.

Un **esempio**:

```c
char buf[PATH_MAX];
// Replace %i with PID, and store the resulting string in buf.
snprintf(buf, PATH_MAX, "/proc/%i/fd", getpid());

DIR *dir = opendir(buf);
struct dirent *dp;
while ((dp = readdir(dir)) != NULL)
{
    if ((strcmp(dp->d_name, ".") != 0) && (strcmp(dp->d_name, "..") != 0))
        printf("\tEntry: %s\n", dp->d_name);
}
closedir(dir);
```

```
user@localhost[~]$ ./program
    Entry: 0 // link to stdin
    Entry: 1 // link to stdout
    Entry: 2 // link to stderr
    Entry: 3 // link to /proc/<PID>/fd directory
```

La funzione *snprintf* è una funzione e consente di salvare una stringa nel registro *buf* di dimensione PATH_MAX. La strina salvata è "*/proc/%i/fd/*" e al posto della *i* viene inserito il *pid* del processo in esecuzione.

Successivamente viene effettuata l'apertura della cartella (*opendir*) presente nel registro *buf* e una lettura degli elementi della cartella stampandoli a video (*readdir*). Gli elementi "." e ".." non vengono stampati, il ciclo *while* continua finché ci sono elementi disponibili alla lettura.

Attenzione! Si ricorda che nel momento in cui un valore entra all'interno della tabella dei file descriptor, viene assegnato *sempre* l'indice più piccolo disponibile.

Nella pratica, questo si può vedere nel momento in cui viene chiuso lo *standard output* (*stdout*):

```c
// We close STDOUT which has FD 1. The remaining file descriptors have
// index 0 (stdin) and 2 (stderr).
close(STDOUT_FILENO);
// We open a new file, to which will be assigned FD 1 automatically
// because it is the lowest available index in the table.
int fd = open("myfile", O_TRUNC | O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
// Printf uses the FD 1, thus, it will print on the file.
printf("ciao\n");
```

La chiusura provoca l'impossibilità di stampa sul terminale poiché non esiste più il file descriptor. Tuttavia, grazie all'apertura di un nuovo file, tramite la *printf*, il sistema scriverà in automatico sul file.

Una delle *system call* utilizzabile per manipolare la file descriptor table è la *dup*. Essa permette di duplicare un file descriptor, o meglio, il valore del file descriptor.

L'utilizzo generico:

```c
#include <unistd.h>

// Returns (new) file descriptor on success, or -1 on error.
int dup(int oldfd);
```

Come unico **argomento** accetta il file descriptor da duplicare.

Un **esempio** per comprendere meglio:

```c
// FDT: [0, 1, 2] -> [0, 2]
close(STDOUT_FILENO);
// FDT: [0, 2] -> [0]
close(STDERR_FILENO);
// FDT: [0] -> [0, 1]
int fd = open("myfile", O_TRUNC | O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
// FDT: [0, 1] -> [0, 1, 2]
dup(1);
// FDT: [0: STDIN, 1: myfile, 2: myfile]
printf("Have a good ");
fflush(stdout);
fprintf(stderr, "day!\n");
```

In questo caso vengono chiusi i due file descriptor riferiti all'output su terminale ($stdout$) e alla gestione degli errori ($stderr$). Successivamente avviene l'apertura del file $myfile$ che prende l'indice di $stdout$, cioè 1, e la duplicazione del contenuto del file descriptor con indice 1, ovvero il file appena aperto. Il duplicato ha come indice il valore 2 nella tabella poiché è l'unico indice più piccolo presente nella tabella.

Tramite la funzione $fflush$ si verifica la corretta scrittura nel file della stringa "$Have\ a\ good$ " e infine si dimostra che stampando il contenuto del file tramite $stderr$, esso è uguale al contenuto presente dentro $stdout$. Questo perché è stata effettuata una duplicazione precedentemente.

2.1 – (*_exit, exit, atexit*) Terminazione

La *system call* *_exit*( ) termina un processo e non ha bisogno di controlli di errori poiché termina sempre con successo.

L'utilizzo generico:

```c
#include <unistd.h>

void _exit(int status);
```

Il primo byte dell'argomento "*status*" definisce lo stato di terminazione del processo. Per convenzione, il valore:

- **Zero** indica che il processo è terminato con **successo**;
- **Non-zero** indica che il processo è terminato **senza successo**, ovvero non è terminato.

Generalmente, i programmatori utilizzano la *system call* *exit*( ) piuttosto che la *_exit*( ).

L'utilizzo generico:

```c
#include <stdlib.h> // N.B. provided by C library

void exit(int status);
```

Il motivo dell'utilizzo di questa *syscall* è dovuta al fatto che essa è inclusa dentro una delle librerie più utilizzate, ovvero "*stdlib*". Inoltre, la libreria definisce anche due macro: EXIT_SUCCESS (0) e EXIT_FAILURE (1).

Quando viene chiamata la *syscall* *exit*( ), durante la terminazione del processo, viene chiamata anche la *syscall* *atexit*.

L'utilizzo generico:

```c
#include <stdlib.h>

// Returns 0 on success, or nonzero on error.
int atexit(void (*func)(void));
```

La *atexit* aggiunge la funzione passata come puntatore ad una lista di funzioni che vengono chiamate nel momento in cui il processo termina.

Attenzione! La funzione *func* deve essere definita di tipo *void*, quindi non deve ritornare valori, e **non** deve accettare argomenti.

Infine, se sono presenti più *atexit*, esse sono chiamate in ordine inverso rispetto a come sono state registrate.

Un **esempio** per chiarire le idee e per capire il concetto di "ordine inverso" di chiusura:

```c
#include <stdlib.h>
#include <stdio.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'output del programma:

```
user@localhost[~]$ ./exit_handlers
    Atexit function 2 called
    Atexit function 1 called
```

Per concludere, un processo può essere terminato anche con l'utilizzo del "*return*" dalla funzione *main*( ):

- Scrivendo "*return n*" equivale a chiamare la *syscall exit*(*n*);
- Scrivendo "*return*" alla fine del *main*( ) è equivalente alla chiamata *exit*(0) presente dal C99 in poi.

2.2 – ($fork, getppid$) Creazione

La *system call* $fork(\ )$ **crea** un nuovo processo, il quale verrà chiamato **figlio** (*child*). Il figlio sarà l'esatta duplicazione del processo chiamate (chiamato **padre** o *parent*) e riceverà tutti i suoi file descriptors e le memorie condivise (approfondimento più avanti).

L'utilizzo generico:

```c
#include <unistd.h>

// In parent: returns process ID of child on success, or -1 on error.
// In created child: always returns 0.
pid_t fork(void);
```

Dopo l'esecuzione della $fork(\ )$, esisteranno due processi nei quali l'esecuzione continuerà dal punto in cui la $fork(\ )$ ritorna. Inoltre, non è possibile determinare quale dei due processi sarà eseguito per primo dalla CPU.

Per capire chi è il padre e chi è il figlio, il sistema operativo controlla il valore di ritorno. Se il valore di ritorno è lo *ID process*, allora esso è il padre che sta creando il figlio; mentre, se il valore di ritorno è zero, allora esso è il figlio. Il sistema operativo può capire che il processo è il padre anche quando il valore di ritorno è -1, tuttavia quest'ultimo indica un errore.

Per **esempio**:

```c
#include <unistd.h>

int main (int argc, char *argv[])
{
    int stack = 111;
    pid_t pid = fork();
    if (pid == -1)
        errExit("fork");
    /* --> Both parent and child come here !!!<-- */
    if (pid == 0)
        stack = stack * 4;
    printf("\t%s stack %d\n", (pid == 0) ? "(child)" : "(parent)", stack);
}
```

Le parti di codice con $pid == 0$ verranno eseguite solo dal figlio poiché esso ritorna il valore zero!

L'output del terminale è:

```
user@localhost[~]$ ./example_fork
    (parent) stack 111
    (child ) stack 444

user@localhost[~]$ ./example_fork
    (child ) stack 444
    (parent) stack 111
```

Questo risultato dimostra che non è possibile determinare quale processo venga eseguito per primo.

Ogni processo ha un padre (*parent*), ovvero colui che lo ha creato tramite la $fork(\ )$. Nel codice del figlio (*child*) è possibile utilizzare la *syscall getppid* per ottenere il *pid* del padre (*parent*) che ha creato il processo.

L'utilizzo generico:

```c
#include <unistd.h>

// Always successfully returns PID of caller's parent.
pid_t getppid(void);
```

La *syscall* **ritorna** sempre un valore valido e mai un errore. Questo è dovuto al fatto che **qualsiasi** processo in esecuzione deriva dal processo *init* con PID = 1. Per cui, se un figlio (*child*) diventa **orfano** (*orphaned*) a causa della terminazione del processo padre (*parent*), allora il figlio (*child*) viene *adottato* dal processo *init*. Questa adozione provoca come risultato il valore 1 alla chiamata della *syscall getppid*.

Per **esempio**:

```c
#include <unistd.h>

int main (int argc, char *argv[])
{
    pid_t pid = fork();
    if (pid == -1)
        errExit("fork");

    if (pid == 0)
        printf("(child ) PID: %d PPID: %d\n", getpid(), getppid());
    else
        printf("(parent) PID: %d PPID: %d\n", getpid(), getppid());

    return 0;
}
```

La stampa sul terminale non è scontata poiché è impossibile determinare chi venga eseguito prima. Quindi, gli scenari possibili sono 3:

1. Il figlio (*child*) viene eseguito **dopo** il padre (*parent*) e il padre non è terminato:

```
(parent) PID: 402 PPID: 350
(child ) PID: 403 PPID: 402
```

2. Il figlio (*child*) viene eseguito **prima** del padre (*parent*):

```
(child ) PID: 403 PPID: 402
(parent) PID: 402 PPID: 350
```

3. Il figlio (*child*) viene eseguito **alla terminazione** del processo padre (*parent*):

```
(parent) PID: 402 PPID: 350
(child ) PID: 403 PPID: 1
```

2.3 – (*wait*, *waitpid*) Monitoraggio di un processo figlio

La *system call wait* permette ad un processo padre (*parent*) di aspettare che il processo figlio (*child*) termini.

L'utilizzo generico:

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

// Returns PID of terminated child, or -1 on error.
pid_t wait(int *status);
```

Se un processo padre **non** ha figli, la *syscall wait* ritorna il valore -1 e alla variabile *errno* viene assegnato il valore ECHILD.

Analogamente, se un processo padre ha un figlio in esecuzione, il processo padre viene bloccato finché il figlio non ha terminato l'esecuzione.

Per **esempio**:

```c
for (int i = 1; i <= 3; ++i)
{
    // Fork and ignore fork failures.
    if (fork() == 0)
    {
        printf("Child %d sleeps %d seconds...\n", getpid(), i);
        // Suspends the calling process for i seconds
        sleep(i);
        _exit(0);
    }
}
pid_t child;
while ((child = wait(NULL)) != -1)
{
    printf("wait() returned child %d\n", child);
}
if(errno != ECHILD)
    printf("(wait) An unexpected error...\n");
```

Vengono creati tre figli grazie al ciclo *for* e successivamente il processo padre entra in uno stato di attesa all'interno del ciclo *while*, il quale stampa il figlio che il padre sta aspettando. Una volta terminati tutti i figli, il valore di *wait* diventa -1 e per terminare correttamente il programma, si controlla anche che il valore -1 non indichi l'errore ECHILD.

Per quanto riguarda i figli, entrano all'interno del costrutto condizionale, stampando la stringa indicando il loro *pid* e il numero di secondo di *sleep*, poi eseguono la funzione *sleep* e al termine "muoiono" con il comando $\_exit(0)$.

Con il seguente output su terminale:

```
user@localhost[~]$ ./example_wait
child 75 sleeps 1 seconds
child 76 sleeps 2 seconds
child 77 sleeps 3 seconds
wait() returned child 75
wait() returned child 76
wait() returned child 77
```

Talvolta, i processi figlio riescono a terminare prima che il padre abbia la possibilità di chiamare la *syscall wait*. In questo particolare caso, il kernel etichetta tali processi come "**processi zombie**". Dunque, per ottimizzare le risorse messe a disposizione, il kernel elimina tali processi **salvando** solamente alcune informazioni essenziali:

- *Process ID*;
- *Termination status*, ovvero il valore ritornato nel momento in cui è terminato;
- *The resource usage statistics*, ovvero le statistiche sull'utilizzo delle risorse.

Analogamente, se un processo padre termina senza chiamare la *wait*, tutti i figli ancora in esecuzione diventano **figli zombie** (*zombie child*) e vengono ***adottati*** dal processo *init*, il quale chiamerà eventualmente in futuro una *wait*.

La *system call waitpid* sospende l'esecuzione del processo chiamante finché un figlio, specificato tramite argomento (*pid*), non cambia il suo stato.

L'utilizzo generico:

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

// Returns a PID, 0, or -1 on error.
pid_t waitpid(pid_t pid, int *status, int options);
```

Gli **argomenti** sono:

- *pid*, ovvero il valore di *pid* del figlio che il padre deve aspettare. I valori possibili:
  - $pid \geq 0$, padre aspetta il figlio avente il *pid* identico al *pid* passato come argomento;
  - $pid = 0$, padre aspetta ogni figlio con lo stesso *process group* del chiamante;
  - $pid < -1$, padre aspetta ogni figlio nel processo di gruppo $|pid|$, ovvero il valore assoluto;
  - $pid = -1$, padre aspetta ogni figlio.
- **status*, valore identico a quello utilizzato per la *wait*.

- *options*, assume uno o più valori (tramite $OR \rightarrow |$) proposti:
  - WUNTRACED: padre esce dallo stato di attesta quando il figlio viene stoppato da un segnale o terminato;
  - WCONTINUED: padre esce dallo stato di attesa quando un figlio riprende la sua esecuzione dopo il segnale SIGCONT;
  - WNOHANG: permette al padre di continuare ad eseguire il suo codice e di non attendere i figli;
  - 0: padre attende solo i figli che terminano e che non si stoppano.

Per **esempio**:

```
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);
```

Il codice crea tre processi e il padre aspetta, tramite la *syscall waitpid*, l'ultimo figlio. Ultimo poiché il valore *pid* viene passato come argomento, ma *pid* come ultimo valore assume il *PID* dell'ultimo processo figlio creato.

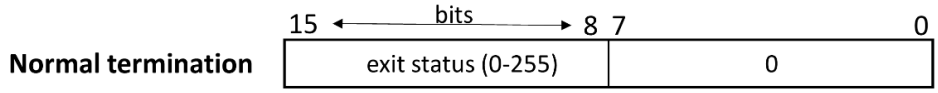
Un altro **esempio**:

```
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);
}
```

Il codice crea un processo figlio e il padre aspetta il figlio finché non viene stoppato o terminato o ripreso da un segnale SIGCONT.

La **lettura del valore *status*** è importante poiché le *syscall wait* e *waitpid* la utilizzano. Il suo scopo è distinguere 4 possibili eventi che potrebbero accadere nei processi figlio:

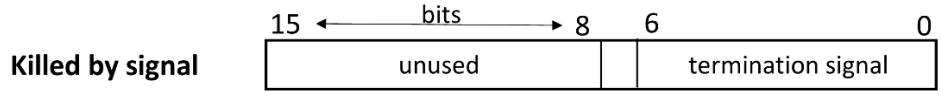
1. Il processo figlio viene terminato con la chiamata della *_exit* o *exit*. La variabile intera *status* è formata da 16 *bit* così divisi:



La macro WIFEXITED ritorna *true* se il processo figlio è uscito normalmente e la macro WEXITSTATUS ritorna il valore "*exit status*" del processo figlio.
Un **esempio**:

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

2. Il processo figlio viene terminato da un segnale. In questo caso, la variabile intera *status* viene divisa nel seguente modo:

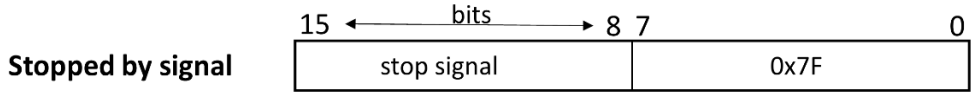


La macro WIFSIGNALED ritorna *true* se il figlio è stato ucciso da un segnale e la macro WTERMSIG ritorna il numero del segnale che ha causato la terminazione del processo.
Un **esempio**:

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

La funzione *strsignal*(*int sig*) appartiene a *string.h* e ritorna una stringa descrivendo il segnale *sig*.

3. Il processo figlio viene stoppato da un segnale. In questo caso, la variabile intera *status* viene divisa nel seguente modo:

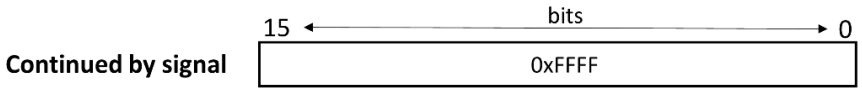


La macro WIFSTOPPED ritorna *true* se il processo figlio è stato stoppato da un segnale e la macro WSTOPSIG(*status*) ritorna il numero del segnale che ha stoppato il processo.

Un **esempio**:

```
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFSTOPPED(status))
    printf("child stopped by signal %d (%s)\n", WSTOPSIG(status),
strsignal(WSTOPSIG(status)));
```

4. Il processo figlio viene ripreso (*resumed*) da un segnale di tipo SIGCONT. In questo caso, la variabile intera *status* viene divisa nel seguente modo:



La macro WIFCONTINUED ritorna *true* se il processo figlio è stato ripreso da un segnale di tipo SIGCONT.
Un **esempio**:

```
waitpid(-1, &status, WUNTRACED | WCONTINUED);
if (WIFCONTINUED(status))
    printf("child resumed by a SIGCONT signal\n");

/* OPPURE */

waitpid(-1, &status, WCONTINUED);
printf("child resumed by a SIGCONT signal\n");
```

3 – Esecuzione del programma (*exec library functions*)

La famiglia di funzioni *exec* sostituisce l'immagine del processo corrente con una nuova immagine del processo.

Le più utilizzate:

```c
#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[]);
```

Attenzione! L'ultimo argomento deve essere il valore NULL per far capire al processo che non deve aspettarsi altri parametri.

Ogni funzione ha:

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

In cui:

- *path*: può essere un *pathname*, ovvero indica il percorso dell'eseguibile, oppure un *filename* che indica il nome di un eseguibile;
- *argv*: una lista o un array di puntatore a stringa che definiscono gli argomenti specificati tramite linea di comando;
- *envp*: un array di puntatori a stringa con il formato "*name* = *value*" che definiscono l'ambiente (*environment*) del programma. Controllare il capitolo per capire meglio.

3.1 – (*execl*)

La funzione *execl* prende il percorso (*path*) del file binario (per esempio "*/bin/ls*") come primo argomento e il comando in chiaro come secondo argomento. L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale.

Per **esempio**:

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

int main (int argc, char *argv[])
{
    execl("/usr/bin/printenv", "printenv", "HOME", (char *)NULL);
    perror("Execl");
}
```

La lista degli argomenti della funzione è:

path (pathname)	:	"/usr/bin/printenv"
argv (list)	:	"printenv", "HOME", (char *)NULL
envp (–)	:	

L'output sul terminale dell'esempio:

```
user@localhost[~]$ ./example_exec
/home/user
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.2 – (*execlp*)

La funzione *execlp*, al contrario della *execl*, prende come argomento il comando eseguibile senza dover specificare l'intero percorso. Questo è possibile solo se il comando è disponibile nella PATH (la *PATH* è una variabile di tipo *environement* nei sistemi *Unix*, *Windows*, ecc. Essa specifica un insieme di cartelle contenenti alcuni eseguibili).

L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale, esattamente come *execl*.

Per **esempio**:

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    execlp("printenv", "printenv", "HOME", (char *)NULL);
    perror("Execl");
}
```

La lista degli argomenti della funzione è:

path (filename)	:	"printenv"
argv (list)	:	"printenv", "HOME", (char *)NULL
envp (–)	:	

L'output sul terminale dell'esempio sarà identico al precedente:

```
user@localhost[~]$ ./example_exec
/home/user
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.3 – (*execle*)

La funzione *execle* ha funzionamento identico alla *execl* ma con la differenza che prende come argomento un array *envp*, ovvero una variabile *environment* avente *NULL* come ultimo elemento.

L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale, esattamente come *execl*.

Per **esempio**:

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    char *env[] =
    {
        "HOME=/home/pippo", (char *)NULL
    };
    execle("/usr/bin/printenv", "printenv", "HOME", (char *)NULL, env);
    perror("Execle");
}
```

La lista degli argomenti della funzione è:

path (pathname)	:	"/usr/bin/printenv"
argv (list)	:	"printenv", "HOME", (char ∗)NULL
envp (array)	:	"HOME =/home/pippo", (char ∗)NULL

L'output sul terminale dell'esempio sarà identico al precedente:

```
user@localhost[~]$ ./example_exec
/home/pippo
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.4 – (*execv*)

La funzione *execv* ha funzionamento identico alla *execl* ma con la differenza che prende come argomento un array *argv* in cui il primo elemento **dovrebbe essere** il percorso (*path*) del file eseguibile e l'ultimo elemento **è** il valore NULL.

L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale, esattamente come *execl*.

Per **esempio**:

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    char *arg[] =
    {
        "printenv",
        "HOME",
        (char *)NULL
    };
    execv("/usr/bin/printenv", arg);
    perror("Execv");
}
```

La lista degli argomenti della funzione è:

path (pathname)	:	"/usr/bin/printenv"
argv (array)	:	"printenv", "HOME", (char *)NULL
envp (–)	:	

L'output sul terminale dell'esempio sarà identico al precedente:

```
user@localhost[~]$ ./example_exec
/home/user
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.5 – (*execvp*)

La funzione *execvp* ha funzionamento identico alla *execv* ma con la differenza che, come primo argomento, è possibile specificare direttamente l'eseguibile senza utilizzare l'intero percorso. Come sempre, questo è possibile grazie al fatto che l'eseguibile è dentro la PATH.

L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale, esattamente come *execl*.

Per **esempio**:

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    char *arg[] =
    {
        "printenv",
        "HOME",
        (char *)NULL
    };
    execvp("printenv", arg);
    perror("Execvp");
}
```

La lista degli argomenti della funzione è:

path (filename)	:	"printenv"
argv (array)	:	"printenv", "HOME", (char *)NULL
envp (–)	:	

L'output sul terminale dell'esempio sarà identico al precedente:

```
user@localhost[~]$ ./example_exec
/home/user
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.6 – (*execve*)

La funzione *execve* ha funzionamento mistro tra *execle* e *execv*. Questa funzione permette di utilizzare una variabile *environment* come in *execle*, ma anche di utilizzare arrays come argomenti come in *execv*.

L'esecuzione della funzione esegue il comando specificato e stampa il risultato nel terminale, esattamente come *execl*.

Per **esempio**:

```c
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv[])
{
    char *arg[] =
    {
        "printenv",
        "HOME",
        (char *)NULL
    };
    char *env[] =
    {
        "HOME=/home/pippo",
        (char *)NULL
    };
    execve("/usr/bin/printenv", arg, env);
    perror("Execve");
}
```

La lista degli argomenti della funzione è:

path (pathname)	:	"/usr/bin/printenv"
argv (array)	:	"printenv", "HOME", (char *)NULL
envp (array)	:	"HOME =/home/pippo", (char *)NULL

L'output sul terminale dell'esempio sarà identico al precedente:

```
user@localhost[~]$ ./example_exec
/home/pippo
```

Il comando *printenv* stampa la *home directory* in cui viene eseguito.

3.7 – Riepilogo caratteristiche funzioni *exec*

È importante ricordare i seguenti punti quando si utilizzando le funzioni della libreria *exec*:

- I parametri di input dei programmi puntano ad un eseguibile;
- Le liste e gli array **devono** essere terminati con un puntatore NULL ((char *)NULL);
- Da convenzione, il primo elemento di *argv* è il nome dell'eseguibile;
- Tutte le funzioni *exec* **non** ritornano un risultato in caso di successo.

Capitolo 3 – Introduzione e fondamenti MentOS

In questo capitolo vengono introdotte le basi del sistema operativo MentOS.

1 – Introduzione a MentOS

MentOS (**Ment**oring **O**perating **S**ystem) è un sistema operativo educativo *open source* e può essere scaricato dalla seguente *repository*: link download.

2.1 – I registri più utilizzati della CPU

Esistono tre tipi di registri:

- $General\text{-}purpose\ registers\ (32\ bit)$

<table>
<tr><td></td><td>AH</td><td>AL</td><td>EAX</td><td>$(16\ bit) \rightarrow$ AX</td></tr>
<tr><td></td><td>BH</td><td>BL</td><td>EBX</td><td>$(16\ bit) \rightarrow$ BX</td></tr>
<tr><td></td><td>CH</td><td>CL</td><td>ECX</td><td>$(16\ bit) \rightarrow$ CX</td></tr>
<tr><td></td><td>DH</td><td>DL</td><td>EDX</td><td>$(16\ bit) \rightarrow$ DX</td></tr>
<tr><td colspan="3"></td><td>ESI</td><td></td></tr>
<tr><td colspan="3"></td><td>EDI</td><td></td></tr>
<tr><td colspan="3"></td><td>EBP</td><td></td></tr>
<tr><td colspan="3"></td><td>ESP</td><td></td></tr>
</table>

Questi registri vengono utilizzati principalmente per salvare risultati di operazioni, operandi, variabili necessari nei cicli come il for, ecc.

- $Segment\ register\ (16\ bit)$

<table>
<tr><td></td><td>CS</td></tr>
<tr><td></td><td>DS</td></tr>
<tr><td></td><td>SS</td></tr>
<tr><td></td><td>ES</td></tr>
<tr><td></td><td>FS</td></tr>
<tr><td></td><td>GS</td></tr>
</table>

Questi registri puntano a segmenti più capienti nella memoria RAM.

- $Status\ and\ control\ registers\ (32\ bit)$

<table>
<tr><td></td><td>EFLAGS</td></tr>
<tr><td></td><td>EIP</td></tr>
</table>

Utilizzati per i permessi dei file, dei processi, ecc.

Più nello specifico, i registri *general-purpose* hanno ognuno una caratteristica specifica:

- EAX: accumulatore di operandi e di risultati;
- EBX: puntatore al segmento DS (*segment register*);
- ECX: contatore per i cicli;
- EDX: puntatori ad I/O;
- ESI: puntatore ai dati puntati dal segmento DS (*segment register*);
- EDI: puntatore ai dati puntati dal segmento ES (*segment register*);
- EBP: puntatore ai dati nello stack (SS *segment register*);
- ESP: puntatore stack, *stack pointer* (SS *segment register*).

Mentre, i registri di *status and control* sono soltanto due e vengono utilizzati per:

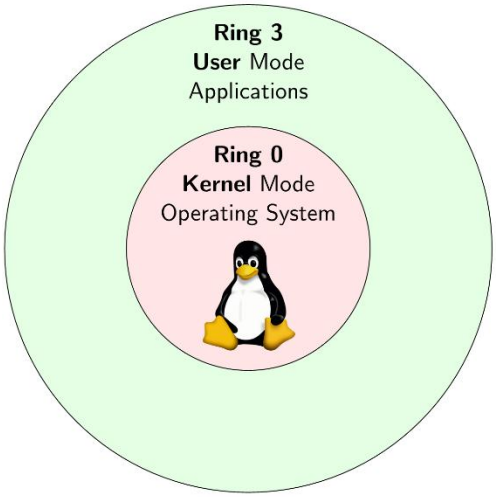
- EIP: puntatore ad istruzioni, conosciuto anche come *program counter*;
- EFLAGS: contiene alcuni bit che sono utili per il sistema e non solo (vedi tabella seguente).

Qui di seguito vengono lasciati alcuni flags:

Bit	Description	Category	Bit	Description	Category
0	Carry flag	Status	11	Overflow flag	Status
2	Parity flag	Status	12-13	Privilege level	System
4	Adjust flag	Status	16	Resume flag	System
6	Zero flag	Status	17	Virtual 8086 mode	System
7	Sign flag	Status	18	Alignment check	System
8	Trap flag	Control	19	Virtual interrupt flag	System
9	Interrupt enable flag	Control	20	Virtual interrupt pending	System
10	Direction flag	Control	21	Able to use CPUID instruction	System

2.2 – Livelli di privilegio

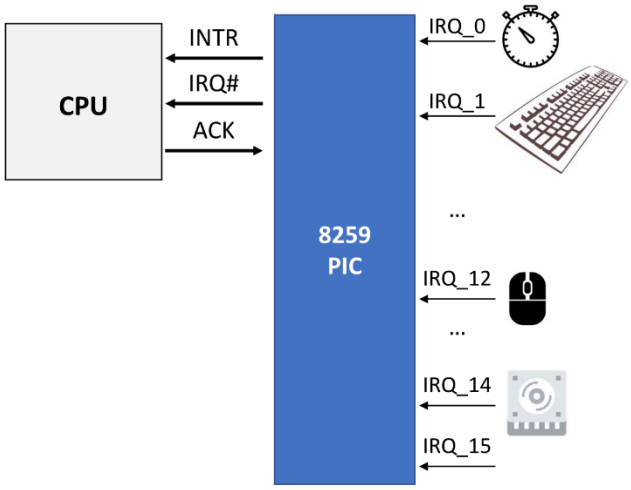
Ci sono **4 livelli** di privilegio che vanno da 0 (**più** privilegi) a 3 (**meno** privilegi):



2.3 – Programmable Interrupt Controller (PIC)

Un PIC è un componente che gestisce le *interrupt*. All'interno di MentOS riesce a gestire fino a 16 *Interrupt Request* (IRQ) numerate da 0 (priorità **alta**) a 15 (priorità **bassa**).

Di default, nella linea IRQ = 0 è presente un timer che funge da sincronizzatore tra le varie operazioni, ovvero un clock dedicato appositamente al sistema operativo.



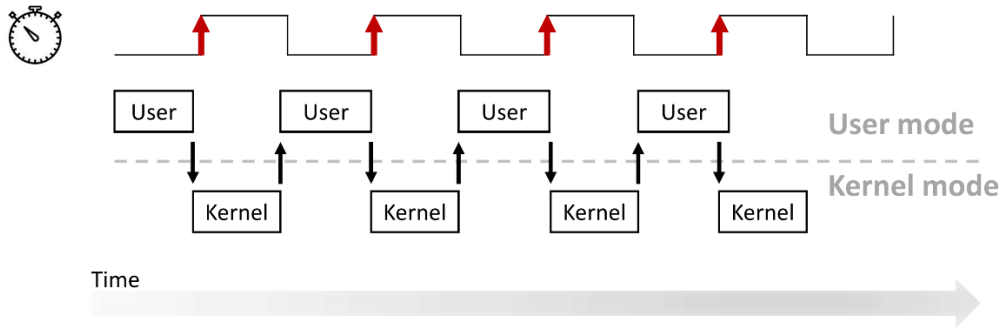
Un esempio di *IRQ*:

1. Pressione di un tasto nella tastiera;
2. PIC alza (mette a 1) il segnale INTR diretto verso la CPU e gli invia la IRQ_1;
3. CPU cambia modalità da *user mode* a *kernel mode* così da gestire la richiesta di interrupt;
4. CPU legge dalla tastiera il tasto premuto;
5. CPU invia come risposta un segnale ACK per notificare IRQ_1 della corretta presa in carico del interrupt;
6. CPU cambia nuovamente modalità tornando alla *user mode*.

2.4 – Timer con IRQ_0

Il Timer è un componente hardware con una frequenza fissa e collegato al *PIC* tramite la linea IRQ_0. *Linux* ha una frequenza del Timer di 100 Hz, ovvero la CPU esegue un processo utente (*user*) per massimo 10 millisecondi. Una volta superati 10 millisecondi, la CPU viene controllata dal *Kernel* per garantire il corretto funzionamento del calcolatore.

Questo discorso è riassumibile con il seguente schema:



3 – Elenco circolare a doppio collegamento (*circular doubly-linked list*)

Il *kernel* di alcuni sistemi operativi mantiene una lista di strutture dati. L'obbiettivo è ridurre l'elevato numero di codice duplicato. Tuttavia, l'implementazione dell'elenco circolare a doppio collegamento, o in gergo *circular doubly-linked list*, porta ad un paio di pro e ad un contro:

Pro:

- Più sicuro e veloce di una propria implementazione ad-hoc;
- Contiene diverse funzioni già scritte.

Contro:

- La manipolazione dei puntatori può essere complessa.

Per utilizzare il meccanismo delle liste, è necessario dichiarare la seguente struttura:

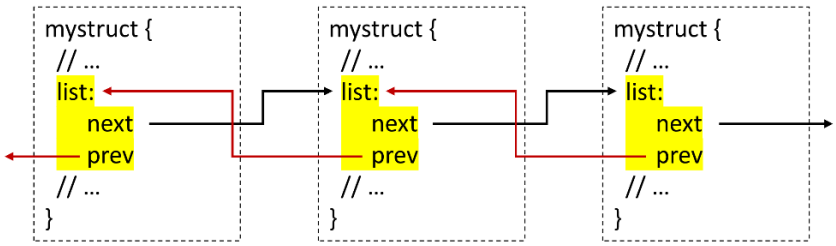
```
typedef struct list_head
{
    struct list_head *next, *prev;
} list_head_t;
```

Questa struttura rappresenta ogni nodo, ovvero *next* punta al nodo successivo e *prev* al nodo precedente.

Data la complessità delle liste, per utilizzarle facilmente è necessario inserire una *list_head* all'interno della struttura che compone la lista. Ovvero:

```
struct mystruct
{
    //...
    list_head_t list;
    //...
};
```

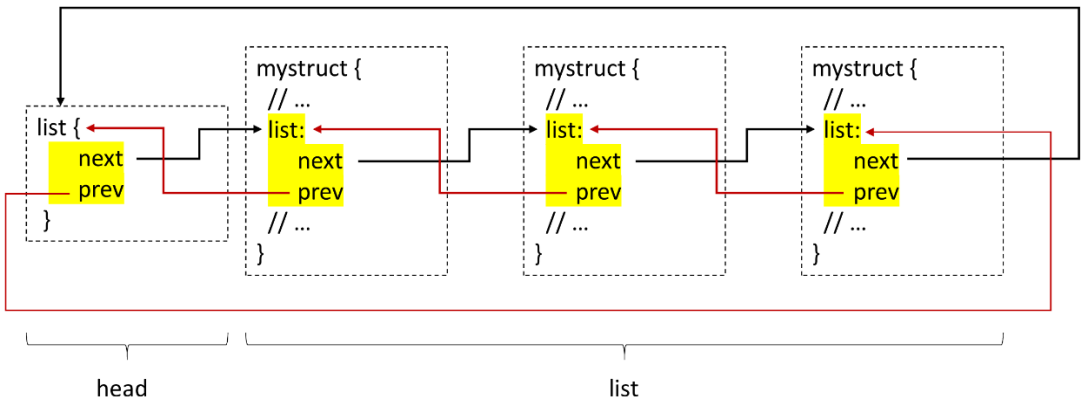
Grazie a questo codice, la struttura *mystruct* può essere collegata per creare una **lista a doppio collegamento** (come in figura):



Nonostante la lista sia diventata più facile da gestire, ancora non è circolare. Per far sì che lo sia, deve esistere una testa, ovvero **deve** essere creata una struttura di tipo *list_head_t*:

```
list_head_t list;
struct mystruct
{
    //...
    list_head_t list;
    //...
};
```

Quindi l'immagine muta e diventa:



Ovvero, una lista a doppio collegamento **circolare**.

Le **funzioni** fornite per essere eseguite con la lista a doppio collegamento circolare sono:

```
list_head_empty(list_head_t *head);
```
La funzione ritorna un valore non-zero se la lista passata come argomento è vuota.

```
list_head_add(list_head_t *new, list_head_t *listnode);
```
La funzione aggiunge il nodo *new* immediatamente dopo il nodo *listnode*.

```
list_head_add_tail(list_head_t *new, list_head_t *listnode);
```
La funzione aggiunge il nodo *new* immediatamente prima il nodo *listnode*.

```
list_head_del(list_head_t *entry);
```
La funzione elimina il nodo *entry* dalla lista.

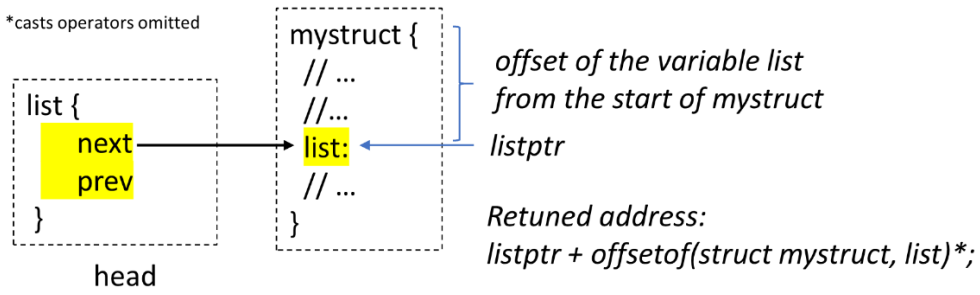
```
list_entry(list_head_t *ptr, type_of_struct, field_name);
```
Ritorna la struttura contenente una *list_head*, ovvero estrae un nodo. In particolare, gli argomenti:

- *ptr* è un puntatore alla *list_head_t*;
- *type_of_struct* è il nome del tipo della struttura (*struct*) contenente una *list_head_t*;
- *field_name* è il nome del puntatore *list_head_t* all'interno della struttura.

Per **esempio**:

```c
// Example showing how to get the first mystruct from a list
list_head_t *listptr = head.next;
struct mystruct *item = list_entry(listptr, struct mystruct, list);
```

Una rappresentazione grafica che potrebbe aiutare:



```
list_for_each(list_head_t *ptr, list_head_t *head);
```
È come un ciclo *forEach*, quindi cicla su tutti i nodi. Come argomenti ha:

- *ptr* è un puntatore libero di tipo *list_head_t*;
- *head* è un puntatore alla testa della lista a doppio collegamento.

Ad ogni invocazione, la variabile *ptr* viene riempita con l'indirizzo del prossimo elemento finché la testa non viene raggiunta.
Per **esempio** di ciclo:

```
list_head_t *ptr;
struct mystruct *entry;
// Inter over each mystruct item in list
list_for_each(ptr, &head)
{
    entry = list_entry(ptr, struct mystruct, list);
    // ...
}
```

Grazie alla funzione *list_entry*, ad ogni iterazione del ciclo si riesce ad estrarre il nodo.

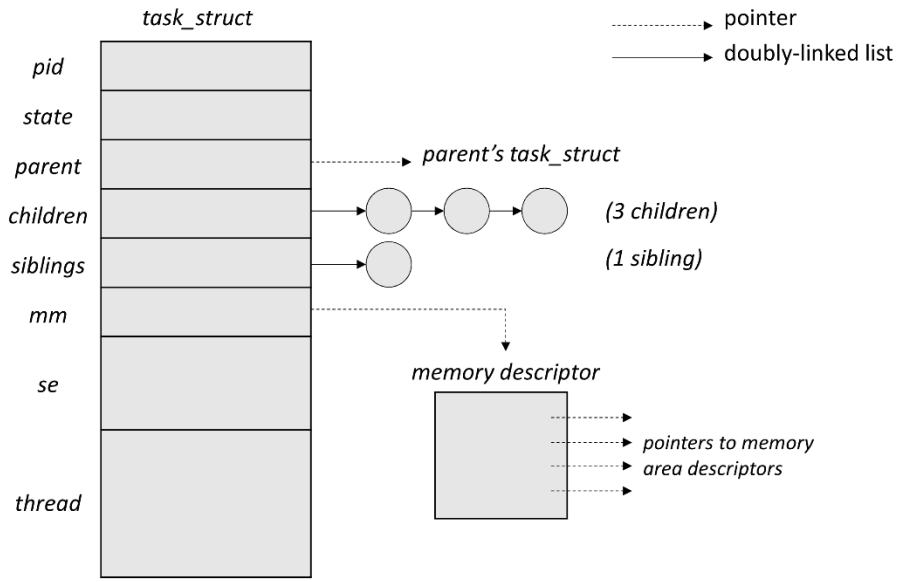
4 – Process descriptor (*task_struct*)

La struttura *task_struct* è utilizzata dal *kernel* per rappresentare un processo e fa parte di una lista circolare a doppio collegamento. Essa è composta in questo modo:

```
struct task_struct
{
    pid_t pid;                    // the process identifier
    unsigned long state;          // the current process's state
    struct task_struct *parent;   // pointer to struct parent process
    struct list_head children;    // head of list of children process
    struct list_head siblings;    // head of list of siblings process
    struct mm_struct *mm;         // memory descriptor
    struct sched_entity se;       // time accounting (aka schedule entity)
    struct thread_struct thread; // context of process
    struct list_head run_list;    // pointer to the process into the scheduler
};
```

N.B. per *siblings* si intendono i processi fratelli.

Una rappresentazione grafica della *task_struct*:



Il numero di figli e fratelli è puramente casuale, questo è solo un esempio.

Nei successivi paragrafi vengono approfonditi i campi della struttura.

4.1 – Process identifier (PID)

Il **process identifier** (PID) è un valore numero che identifica un processo. Quando un nuovo processo viene creato, un nuovo PID viene generato sommando 1 all'ultimo PID assegnato.

Nei sistemi operativi *Linux*, il massimo valore che PID può assumere è 32768. Quando si raggiunge tale valore, prima di assegnare un nuovo PID, l'ultimo PID assegnato viene resettato e quindi posto a 0.

La macro RESERVED_PID, solitamente posta a 300, viene utilizzata per riservare alcuni PID ai processi di sistema e *daemons* (?). Tutti i processi degli utenti hanno un PID maggiore rispetto al RESERVED_PID.

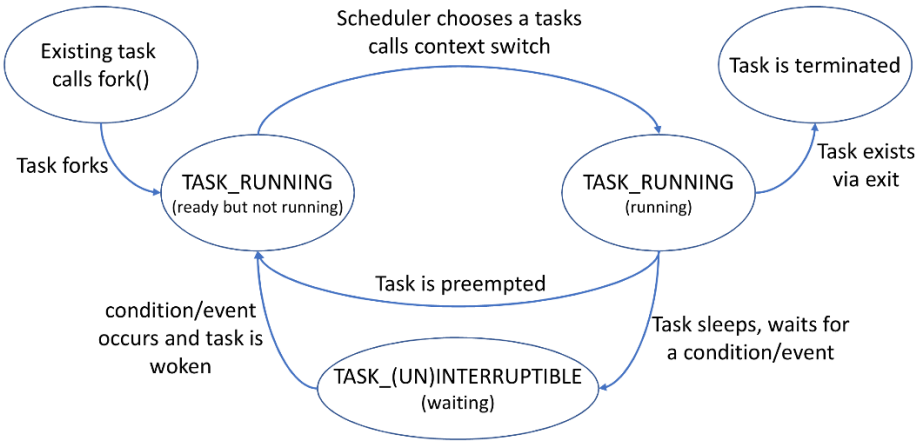
4.2 – Stato di un processo

Lo stato di un processo è un valore numero che descrive lo stato corrente del processo. Nella struttura $task\_struct$, lo stato è rappresentato dalla variabile $state$. I possibili stati sono:

- TASK_RUNNING: indica che il processo è in esecuzione o che ha tutte le risorse pronte per essere eseguite ma ancora non è nella CPU;
- TASK_INTERRUPTIBLE: il processo è bloccato ($sleep$) e aspetta qualche condizione (un segnale) per riprendere l'esecuzione. Se la condizione esiste, e ovviamente se si manifesta, il $kernel$ cambia lo stato del processo in TASK_RUNNING. Alcuni esempi di segnali che modificano la condizione: $interrupt$, segnali generici, risorse rilasciate;
- TASK_UNINTERRUPTIBLE: questo stato è identico a TASK_INTERRUPTIBLE ma la condizione di attesa non termina con un segnale specifico. Il processo deve aspettare una chiamata specifica, per esempio il processo è in attesa che i dati vengano trasferiti;
- TASK_STOPPED: indica che un processo in esecuzione è stato fermato; quindi, l'attività non è né in esecuzione né idonea/pronta per l'esecuzione;
- EXIT_ZOMBIE: l'esecuzione del processo è terminata, ma il processo padre ($parent$) non ha ancora eseguito una $system\ call$ (per esempio $wait4(0)$ o $waitpid(\ \ )$) per ottenere le informazioni riguardo la morte del figlio ($child\ process$);
- EXIT_DIED: è l'ultimo stato. Il processo viene rimosso dal sistema perché il processo padre ($parent$) ha appena ricevuto una $system\ call$ riguardo le informazioni del figlio (per esempio $wait4(0)$ o $waitpid(\ \ )$).

Il processo $init$ ha PID $= 1$.

Uno schema per rappresentare il percorso che fanno i processi:



4.3 – Relazioni tra processi

I processi creati da un programma hanno varie relazioni tra padri (*parents*) e figli (*children*). Quando un processo padre (*parent*) crea molteplici figli (*children*), quest'ultimi hanno una relazione di **fratellanza**.

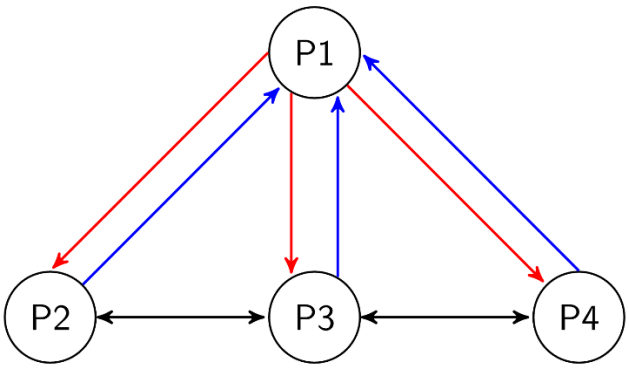
Nella struttura *tasl_struct*, le relazioni tra processi riguardano i seguenti campi:

```
struct task_struct
{
    // ...
    pid_t pid;                    // the process identifier
    struct task_struct *parent;   // pointer to parent process
    struct list_head children;    // list of children process
    struct list_head siblings;    // list of siblings process
    // ...
};
```

Escluso il campo *pid* che è stato descritto nei paragrafi precedenti, i campi *parent*, *children*, *sibling* vengono definiti qui di seguito:

- **parent* → puntatore al processo padre (*process's parent*);
- *children* → la testa della lista che contiene tutti i figli creati dal processo;
- *sibling* → la testa della lista che contiene tutti i figli creati dal processo padre.

È possibile rappresentare le relazioni tra quattro processi tramite una figura:



La linea **rossa** va dal **padre** → **figlio**.

La linea **blu** va dal **figlio** → **padre**.

La linea **nera** mostra le **relazioni tra fratelli**.

4.4 – Gestione del tempo

Nella struttura *task_struct*, il campo:

```
struct task_struct
{
    // ...
    struct sched_entity se;      // time accounting (aka schedule entity)
    // ...
};
```

Indica la priorità e il tempo di esecuzione di un processo.

In particolare, la struttura *sched_entity* è formata nel seguente modo:

```
struct sched_entity
{
    int    prio;             // priority
    time_t start_runtime;    // start execution time
    time_t exec_start;       // last context switch time
    time_t sum_exec_runtime; // overall execution time
    time_t vruntime;         // weighted execution time
};
```

I campi in questo tipo di struttura sono:

- *prio*: Indica la priorità dell'esecuzione di un processo. È un valore compreso tra 100 e 139 (inclusi), in cui 100 indica la priorità più alta possibile e 139 indica la priorità più bassa possibile.
  Di default la priorità di un nuovo processo è di 120.
  Un processo può aumentare/decrementare il suo *prio* usando la *system call* $nice(inc)$, la quale prende come argomento un valore compreso tra -20 e 19 (inclusi).
  Per esempio:
  - $nice(1)$: incrementa il valore *prio* del processo chiamante di 1 unità, quindi passando da 120 a 121
  - $nice(-5)$: decrementa il valore *prio* del processo chiamante di 5 unità, quindi passando da 120 a 115
- *start_runtime*: Indica il tempo di esecuzione del sistema quando il processo è stato eseguito per la primissima volta nella CPU
- *exec_start*: Indica il tempo di esecuzione del sistema dell'ultima esecuzione del processo nella CPU

- *sum_exec_runtime*: Indica il tempo totale di esecuzione speso dal processo all'interno della CPU
- *vruntime*: Indica il *virtual runtime*, ovvero il tempo di esecuzione *ponderato* generale trascorso dal processo nella CPU

4.5 – Contesto di un processo

Nella struttura *task_struct*, il campo:

```
struct task_struct
{
    // ...
    struct thread_struct thread; // context of process
};
```

Indica il contenuto di un processo ogniqualvolta esso viene disattivato. Precisamente, indica i valori presenti nei registri nel momento in cui il processo è stato disattivato.

In particolare, la struttura *thread_struct* contiene i seguenti campi:

```
struct thread_struct
{
    uint32_t ebp;         // base pointer register
    uint32_t esp;         // stack pointer register
    uint32_t ebx;         // base register
    uint32_t edx;         // data register
    uint32_t ecx;         // counter
    uint32_t eax;         // accumulator register
    uint32_t eip;         // Instruction Pointer Register
    uint32_t eflags;      // flag register
    bool_t fpu_enabled;   // is FPU enabled?
    savefpu fpu_register; // FPU context
};
```

L'utilizzo dei registri è stato già ampiamente spiegato ad inizio capitolo.

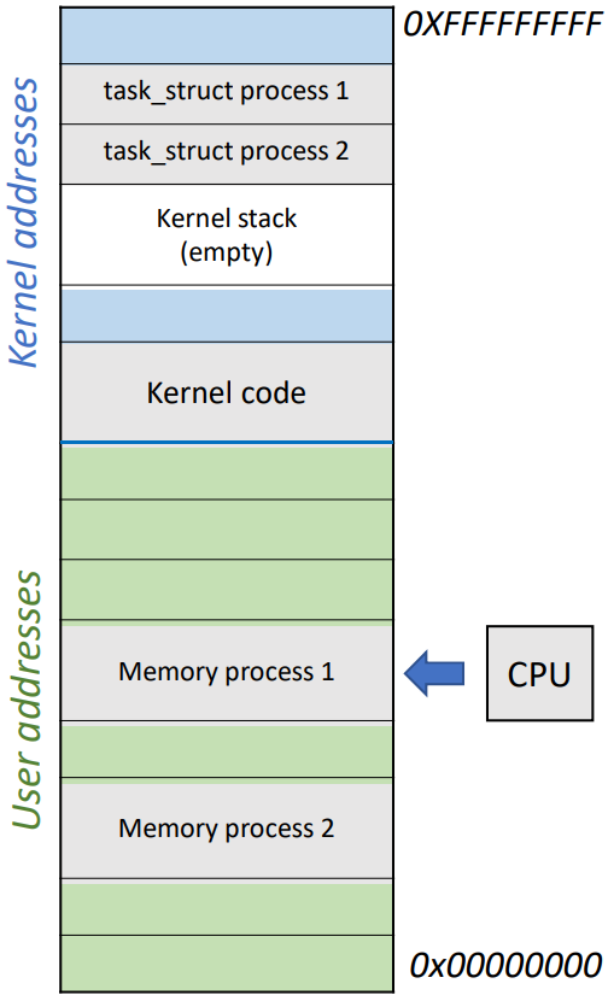
Questa struttura diventa fondamentale durante un *context switch*, ovvero quando il sistema operativo passa da *user mode* a *kernel mode*. Si guardi l'esempio nel paragrafo successivo per comprendere meglio.

4.6 – Cambio di contesto (*context switch*)

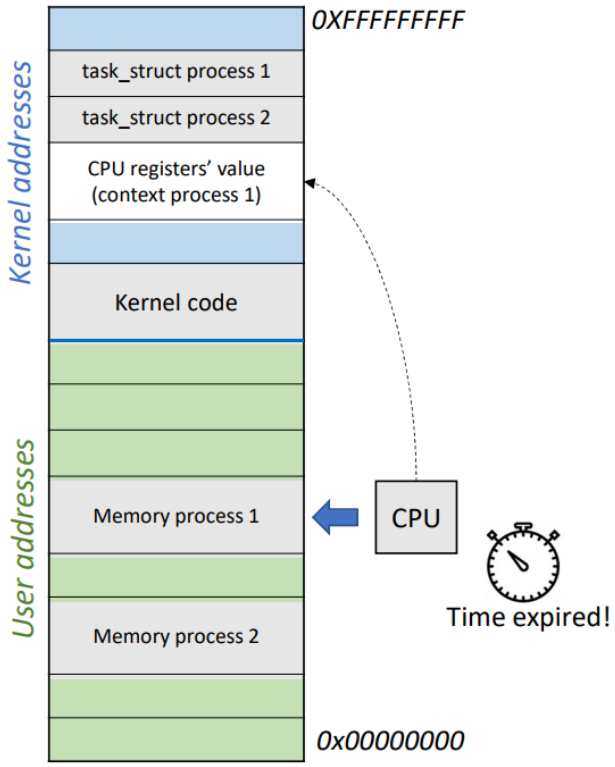
La CPU esegue il cambio di contesto (*context switch*) per cambiare il processo eseguito.

Il seguente esempio mostra i passi effettuati dal sistema operativo per salvare il processo corrente (*process* 1) situato nella *User addresses*, e per riprendere l'esecuzione del processo precedentemente fermato (*process* 2) situato nella *Kernel addresses*.

La **situazione iniziale** è la seguente:

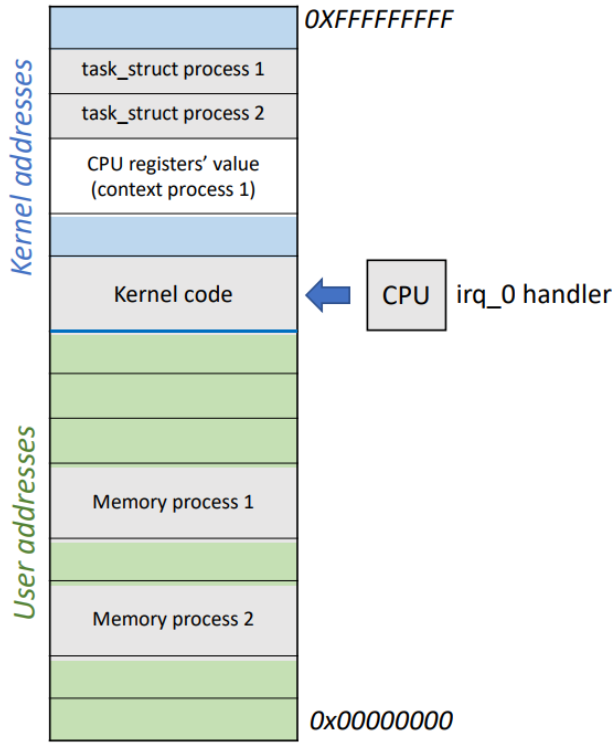


Prima di iniziare, si specifica che la parte in verde rappresenta l'area dedicata ai processi per l'utente e la parte in azzurro l'area dedicata la *kernel*, ovvero all'incirca 1/3.

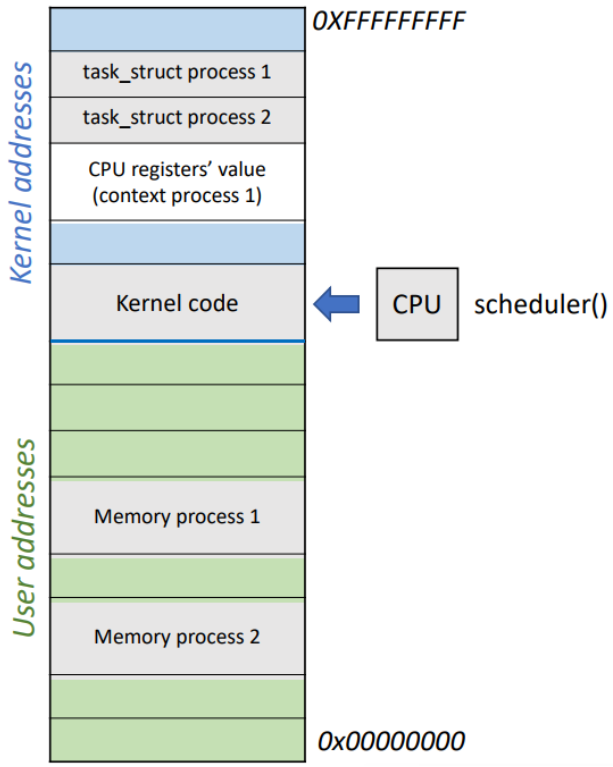


(**1**) Quando il tempo esaurisce, il sistema operativo restituisce il controllo della CPU al *kernel*.
Sulla linea IRQ_0 si presenta un interrupt, il PIC (*Programmable Interrupt Controller*) presenta il segnale INTR alla CPU.

Quando il segnale INTR si presenta, la CPU passa dallo stato numero 3 (*user mode*) allo stato numero 0 (*kernel mode*).
Una volta cambiati i privilegi, i valori presenti nei registri della CPU vengono momentaneamente allocati nello *stack* del *kernel* grazie al "contesto di un processo" visto nel paragrafo precedente.

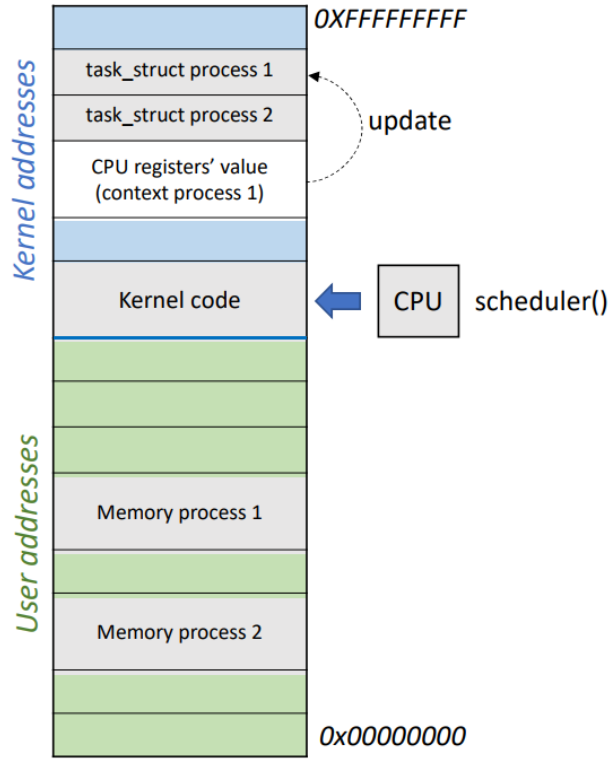


(**2**) La CPU esegue comodamente la richiesta di interrupt ricevuta dalla linea IRQ_0.

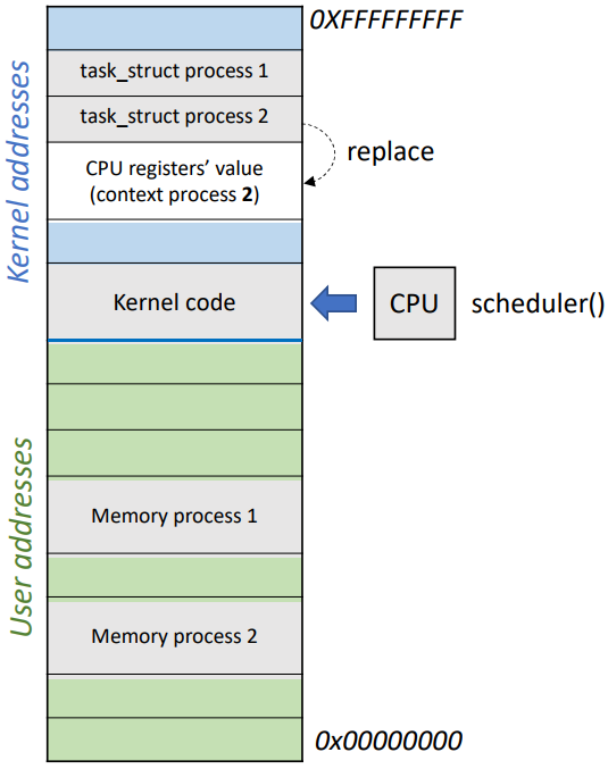


(**3**) Lo *scheduler* viene chiamato per aggiornare le variabili che conteggiano il tempo del processo interrotto e per selezionare il prossimo processo da eseguire.

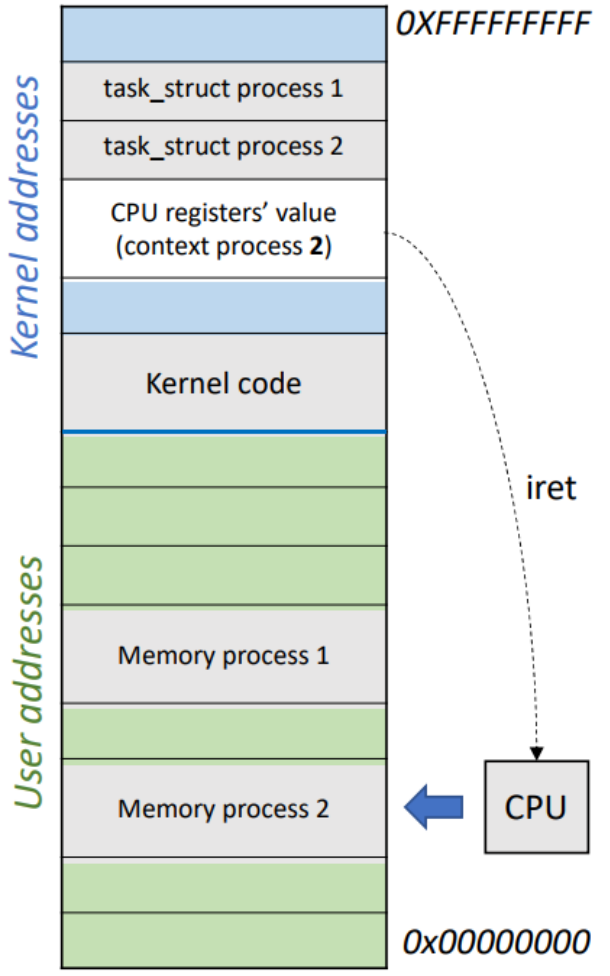
In questo esempio, lo *scheduler* seleziona il processo 2 come il prossimo.



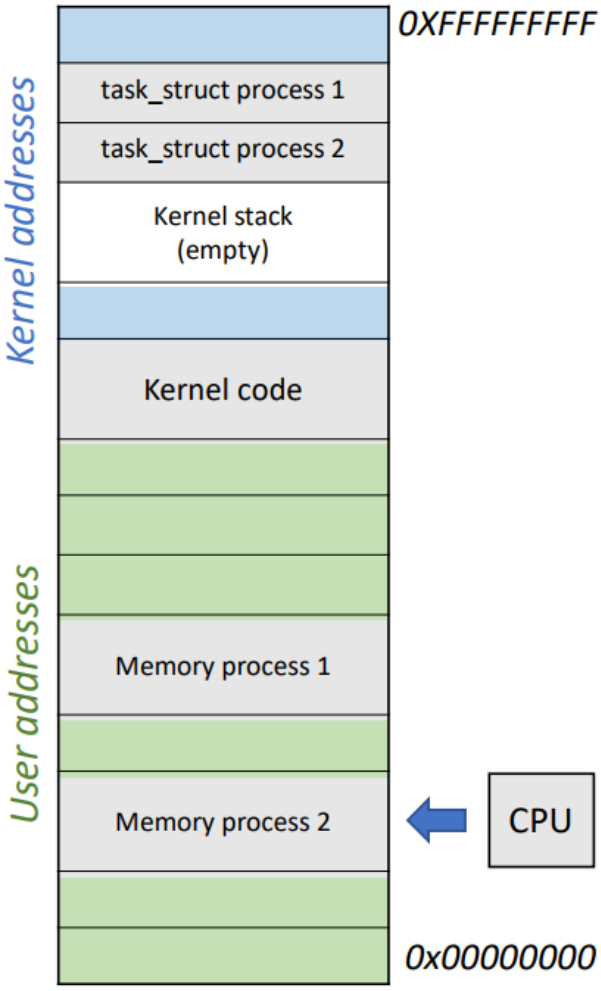
(**4**) Il *kernel* aggiorna la struttura *thread_struct* del processo numero 1 tramite la sua struttura di tipo *task_struct*.
Facendo così salva momentaneamente i registri della CPU quando era in esecuzione il processo numero 1.



(**5**) Il *kernel* sovrascrive i registri della CPU inserendo i valori dei registri del processo numero 2 nell'area di memoria riservata alla "comunicazione" dei registri alla CPU.



(**6**) Il *kernel* sposta i valori dal suo *stack* ai registri della CPU ed esegue l'istruzione assembly *iret*.
Quest'ultima cambia i privilegi della CPU dallo stato 0 (*kernel mode*) allo stato 3 (*user mode*).



(**7**) Il contenuto del processo numero 2 è finalmente all'interno della CPU.
La CPU esegue il codice del processo numero 2 in modalità *user* finché non avviene un altro cambio di contesto (*context switch*).

5.1 – Strutture dati

La struttura dati *runqueue* è la struttura più importante dello *scheduler*. Essa contiene tutti i processi di sistema che si trovano nello stato di esecuzione (*running state*):

```
struct runqueue
{
    unsigned long nr_running; // number of processes in running state
    struct task_struct *curr; // pointer to current running process
    struct list_head_t queue; // head of list of processes in running state
};
```

Attenzione! Il campo *queue* indica la testa di una lista circolare a doppio collegamento contenente tutti i processi di sistema che sono in stato di esecuzione. Di conseguenza, il campo *run_list* di tipo *list_head*, ovvero una struttura, viene aggiunto alla struttura *task_struct*.

Lo ***scheduler*** viene chiamato dopo la gestione di un *interrupt* o di una eccezione (*exception*), ovvero quando è necessario selezionare il prossimo processo da eseguire. In particolare, le operazioni effettuate dallo *scheduler* sono le seguenti:

1. Aggiorna il tempo delle varie variabili dei processi;
2. Prova a "svegliare" un processo in attesta. Ogniqualvolta una condizione di attesta del processo viene incontrata, lo *scheduler* cambia il suo stato in "*running*" e lo inserisce dentro una coda di esecuzione (*runqueue*);
3. Esegue l'algoritmo di *scheduling* per selezionare il prossimo processo da eseguire nella CPU. I processi vengono selezionati dalla coda di esecuzione (*runqueue*);
4. Esegue il cambio di contesto, ovvero *context switch*.

5.2 – Algoritmi di scheduling

La funzione *pick_next_task* viene chiamata dallo scheduler per acquisire il prossimo processo da eseguire. Per scegliere il prossimo processo da eseguire, esistono diversi algoritmi; il sistema operativo MentOS ne implementa tre: *Round-Robin* (**RR**), *Highest Priority First* (**Priority**), *Completely Fair Scheduler* (CFS).

Attenzione! Nei seguenti algoritmi sarà presente una variabile chiamata "*next*" la quale è dichiarata esternamente dei codici presentati. Essa è un puntatore di tipo *task_struct* e viene ritornata alla fine con: "return next".

5.2.1 – Round-Robin

L'algoritmo di scheduling *Round-Robin* assegna circolarmente ad ogni processo di sistema un intervallo di tempo fisso. È semplice da implementare, semplice da comprendere ed evita il fenomeno di *starvation* (ovvero una permanenza infinita del processo all'interno della lista dei processi in attesa).

Il codice scritto nel linguaggio C e funzionante per il sistema operativo MentOS è il seguente:

```
1 // get the next process after the current one
2 list_head *nNode = runqueue->curr->run_list.next;
3
4 // check if we reached the head of list_head
5 if (nNode == &runqueue->queue)
6     nNode = nNode->next;
7
8 // get the task_struct
9 next = list_entry(nNode, struct task_struct, run_list);
```

La spiegazione è piuttosto semplice. Viene creato un puntatore di nome *nNode* di tipo *list_head* e inizializzato accedendo alla struttura *runqueue* (contenente tutti i processi di sistema), successivamente al campo *curr* (puntatore al processo attualmente in esecuzione) che è di tipo *task_struct*, ovvero la struttura utilizzata dal kernel per rappresentare un processo; in questa struttura, si accede al campo *run_list* (puntatore al processo dentro lo scheduler) di tipo *list_head* nella quale ci sono solamente i due campi *next*, per accedere al processo successivo, e *prev*, per accedere al processo precedente.

Una volta salvato il processo successivo a quello in esecuzione, si verifica che sia la testa accedendo all'indirizzo (& davanti al nome) ed entrando nel campo *queue*, ovvero la testa della lista dei processi che sono nello stato di esecuzione. Nel caso in cui sia la testa, si riassegna al puntatore il processo immediatamente successivo alla testa tramite il campo *next*.

Per concludere l'algoritmo, si assegna alla variabile *next* la struttura estratta, o in altri termini il nodo della lista puntato da *nNode*.

5.2.2 – Highest Priority First

L'algoritmo di scheduling *Highest Priority First* (abbreviato in *Priority*) associa una priorità ad ogni processo e viene assegnato alla CPU il processo più alto. Tralasciando gli aspetti teorici di come possa venir calcolata la priorità, l'aspetto più interessante è che l'implementazione soffre di un problema grave: la **starvation** (*attesa infinita*).

Il codice scritto nel linguaggio C e funzionante per il sistema operativo MentOS è il seguente:

```
1  // get the first element of the list
2  next = list_entry(runqueue->queue.next, struct task_struct, run_list);
3
4  // Get its static priority.
5  time_t min = next->se.prio;
6
7  list_head *it;
8  // Inter over the runqueue to find the task with the smallest priority value
9  list_for_each (it, &runqueue->queue) {
10     task_struct *entry = list_entry(it, struct task_struct, run_list);
11     // Check entry has a lower priority
12     if ((entry->se.prio) <= (min)) {
13         min = entry->se.prio;
14         next = entry;
15     }
16 }
```

La variabile *next* acquisisce la testa della coda dei processi e la variabile *min* la sua priorità (N.B. la variabile *min* poteva essere di tipo *int* dato che la priorità, ovvero il campo *prio*, è un valore intero).

Il ciclo parte dalla testa e confronta ogni priorità del processo (riga 11) con la priorità più piccola trovata, inizialmente la priorità del processo *next*. Nel caso in cui si trovi un valore uguale o minore, la variabile *min* viene aggiornata con il nuovo valore e il puntatore *next* acquisisce il nuovo processo con priorità minore.

Per provocare e dunque verificare il fenomeno di *starvation*, basta utilizzare il comando *nice* di MentOS con valori positivi. In questo modo, viene diminuita la priorità del processo (il processo Shell) visto che l'unico processo rimanente è il processo Init e visto che Init non terminerà mai, verrà eseguito solo Init, senza mai ridare il controllo al processo Shell, dando l'impressione che il sistema sia bloccato.

5.2.3 – Completely Fair Scheduler

L'algoritmo di scheduling *Completely Fair Scheduler* (abbreviato in *CFS*) ha l'obbiettivo di prevenire la *starvation* assegnando la CPU equamente a tutti i processi del sistema.

Come si vedrà di seguito dall'implementazione in C, l'algoritmo si basa sul campo *vruntime*, ovvero una sorta di peso:

```
1  // Get the weight of the current process.
2  // (use GET_WEIGHT macro!)
3  int weight = GET_WEIGHT(runqueue->curr->se.prio);
4
5  if (weight != NICE_0_LOAD) {
6      // get the multiplicative factor for its delta_exec.
7      double factor = NICE_0_LOAD / weight;
8
9      // weight the delta_exec with the multiplicative factor.
10     delta_exec = delta_exec * factor;
11 }
12
13 // Update vruntime of the current process.
14 runqueue->curr->se.vruntime += delta_exec;
15
16 // Get the first element of the list
17 next = list_entry(runqueue->queue.next, struct task_struct, run_list);
18
19 // Get its virtual runtime.
20 time_t min = next->se.vruntime;
21
22 // Inter over the runqueue to find the task with the smallest vruntime value
23 list_head *it;
24 list_for_each (it, &runqueue->queue) {
25     task_struct *entry = list_entry(it, struct task_struct, run_list);
26     // Check entry has a lower vruntime
27     if ((entry->se.vruntime) < (min)) {
28         min = entry->se.vruntime;
29         next = entry;
30     }
31 }
```

La macro *GET_WEIGHT* restituisce il peso relativo alla priorità fornita. Se il peso è diverso dal peso di una *task* con priorità normale, cioè 1024, (NICE_0_LOAD) viene aggiornato il peso del *vruntime* con *delta_exec* (ovvero il tempo speso nella CPU dal processo l'ultima volta).

Infine, riparte il controllo con il ciclo per verificare il processo con il peso minore. La *starvation* viene evitata poiché appena viene trovato un processo con peso minore, ad esso gli viene assegnata la CPU.

Capitolo 4 – System V IPC, semafori e segnali

1 – Introduzione a "System V IPC"

Per **System V** (chiamato anche system five o SysV) si intende una delle prime versione commerciali del sistema operativo $Unix$. Venne originariamente sviluppato da AT&T e rilasciato nel 1983. Dal momento della sua uscita fino ad adesso sono uscite 4 versioni: $1, 2, 3$ e 4.

Per **I**nter**p**rocess **C**ommunication (**IPC**) ci si riferisce al meccanismo che coordina le attività tra i processi cooperanti. Un esempio comune è la gestione degli accessi alle risorse del sistema.

Quindi, i System V IPCs fanno riferimento a tre differenti meccanismi per la comunicazione tra processi:

- **Semafori**, consentono ai processi di sincronizzare le loro azioni. Un semaforo è un valore mantenuto nel kernel ed è opportunamente modificato dai processi di sistema prima di effettuare operazioni critiche.
- **Coda di messaggi** ($Message\ queues$), può essere usata per passare messaggi tra processi.
- **Memoria condivisa** ($Shared\ memory$), consente a più processi di condividere una loro regione di memoria

Altri IPC che verranno approfonditi:

- $Signals$
- $Pipes$
- $FIFOs$

1.1 – Creazione e apertura di un oggetto "System V IPC"

Ogni meccanismo System V IPC ha associato una system call di tipo *get* (msgget, semget o shmget), le quali sono analoghe alla system call *open*.

Data una chiave intera *key* (analoga al filename), la syscall *get* può:

- O creare un nuovo IPC e poi ritornare il suo identificatore univoco;
- O ritornare l'identificatore di un IPC esistente.

Un identificatore IPC (IPC *identifier*) è l'analogo ad un *file descriptor*. È utilizzato per far riferimento agli oggetti IPC.

Un esempio di **creazione** di un **semaforo**:

```
// PERM: rw-------
id = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1)
    errExit(semget);
```

Come con tutte le chiamate *get*, la chiave *key* è il primo argomento. Il valore di ritorno della syscall è un identificatore IPC (*identifier IPC*) che è un codice identificativo unico rappresentante l'oggetto IPC nel sistema.

Le **chiavi System V IPC** sono valori interi rappresentati tramite il tipo di dato *key_t*. L'IPC traduce una chiave in un corrispondente identificatore IPC (*identifier IPC*).

Per **evitare conflitti** con chiavi già esistenti, è possibile, durante la creazione di un nuovo oggetto IPC, inserire il flag IPC_PRIVATE. Così facendo il problema di trovare una chiave unica verrà affidato al kernel.

Un esempio di utilizzo:

```
id = semget(IPC_PRIVATE, 10, S_IRUSR | S_IWUSR);
```

Questa tecnica è molto **utile** nei casi di applicazioni multiprocessore dove il processo padre crea l'oggetto IPC prima di eseguire una *fork*( ), con il risultato che il figlio eredita l'identificatore dell'oggetto IPC.

Esiste un terzo modo per generare chiavi, ovvero l'utilizzo della **syscall** *ftok* (file to key). Questa funzione converte un *pathname* e un *proj_id* in una chiave System V IPC.

Il codice per farlo:

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

// Returns integer key on success, or -1 on error (check errno)
key_t ftok(char *pathname, int proj_id);
```

Ovviamente il *pathname* fornito come argomento deve riferirsi ad un file esistente ed accessibile. Tipicamente *pathname* si riferisce ad un file o ad un dizionario creato dall'applicazione.

Gli ultimi 8 bit di *proj_id* vengono utilizzati, per cui non devono essere uguali a zero.

Un **esempio** di utilizzo della funzione *ftok*:

```
key_t key = ftok("/mydir/myfile", 'a');
if (key == -1)
    errExit("ftok failed");

int id = segmet(key, 10, S_IRUSR | S_IWUSR);
if (id == -1)
    errExit("segmet failed");
```

Il carattere '*a*' ha come codice ASCII il valore 097 e come codice binario 01100001.

1.2 – Struttura dati (ipc_perm)

Il kernel mantiene una struttura dati per ogni istanza di oggetto System V IPC. Ogni struttura dati associata include la sottostruttura *ipc_perm* che detiene i permessi concessi.

La struttura è la seguente:

```
struct ipc_perm
{
    key_t __key;            /* Key, as supplied to 'get' call */
    uid_t uid;              /* Owner's user ID */
    gid_t gid;              /* Owner's group ID */
    uid_t cuid;             /* Creator's user ID*/
    gid_t cgid;             /* Creator's group ID*/
    unsigned short mode;    /* Permissions */
    unsigned short __seq;   /* Sequence number */
};
```

Il campo *key* è già stato spiegato nel paragrafo precedente.

Il campo *uid* e *gid* specificano la proprietà dell'oggetto IPC.

Il campo *cuid* e *cgid* contengono lo *user* e l'ID del gruppo, del processo che ha creato l'oggetto. Questi campi sono **immutabili**, cioè non possono essere modificati!

Il campo *mode* contiene le *permissions mask* per l'oggetto IPC, ovvero una serie di bit (minore di 9 bit) che identificano i permessi. Attenzione che gli oggetti IPC possono essere **solo eseguiti o letti** (*Read and Writing*), quindi un ipotetico permesso di esecuzione verrà ignorato per esempio.

Un **esempio** che mostra il tipico utilizzo del *semctl* per cambiare il proprietario di un semaforo:

```
struct semid_ds semq;
// get the data structure of a semaphore from the kernel
if(semctl(semid, 0, IPC_STAT, &semq) == -1)
    errExit("semctil get failed");
// change the owner of the semaphore
semq.sem_perm.uid =newuid;
// update the kernel copy of the data structure
if (semctl(semid, IPC_SET, &semq) == -1)
    errExit("semctil set failed");
```

In modo similare, le chiamate di sistema *shmctl* e *msgctl* sono utilizzate per aggiornare la struttura dati del kernel di una *shared memory* e di una *message queue*.

2.1 – (ipcs)

Usando il comando *ipcs*, è possibile ottenere informazioni sugli oggetti IPC presenti nel sistema. Di default, vengono mostrati tutti gli oggetti, come nel seguente esempio:

```
user@localhost[~]$ ipcs
------ Message Queues --------
key    msqid      owner     perms     used-bytes  messages
0x1235 26         student   620       12          20

------ Shared Memory Segments --------
key    shmid      owner     perms     bytes     nattch    status
0x1234 0          professor 600       8192      2

------ Semaphore Arrays --------
key    semid      owner     perms     nsems
0x1111 102        professor 330       20
```

2.2 – (ipcrm)

Usando *ipcrm*, è possibile rimuovere gli oggetti IPC dal sistema:

Remove a message queue:

```
ipcrm -Q 0x1235   ( 0x1235 is the key of a queue )
ipcrm -q 26       ( 26 is the identifier of a queue )
```

Remove a shared memory segment

```
ipcrm -M 0x1234   ( 0x1234 is the key of a shared memory seg. )
ipcrm -m 0        ( 0 is the identifier of a shared memory seg. )
```

Remove a semaphore array

```
ipcrm -S 0x1111   ( 0x1111 is the key of a semaphore array )
ipcrm -s 102      ( 102 is the identifier of a semaphore array )
```

3.1 – Creazione e apertura dei semafori (semget, semctl, semun)

La chiamata di sistema ***semget*** crea un nuovo **insieme di semafori** oppure restituisce l'identificatore di uno esistente, ma solitamente viene utilizzato per il primo scopo.

Il codice identificativo:

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

// Returns semaphore set identifier on success, or -1 error
int semget(ket_t key, int nsems, int semflg);
```

Gli argomenti sono:

- ***key***, la chiave IPC che può essere generata in tre modi (vedere il paragrafo 1.1)
- ***nsems***, specifica il numero di semafori in quell'insieme e deve essere maggiore di zero
- ***semflg***, è una bitmask, ovvero una serie di bit che specificano i permessi da assegnare ad un nuovo insieme di semafori o per controllare quelli esistenti in un insieme. I permessi ammessi sono quelli della syscall *open* (tenendo conto delle eccezioni come specificato nel paragrafo 1.2) e in aggiunta:
    - **IPC_CREAT**: se non esiste alcun insieme di semafori con la chiave specificate, viene creato un nuovo insieme;
    - **IPC_EXCL**: usata insieme a alla IPC_CREAT, essa fa fallire la *semget* nel caso in cui l'insieme di semafori esista di già con la chiave specificata.

Un **esempio** di un insieme di 10 semafori:

```c
int semid;
ket_t key = //... (generate a key in some way, i.e. with ftok)

// A) delegate the problem of finding a unique key to the kernel
semid = semget(IPC_PRIVATE, 10, S_IRUSR | S_IWUSR);

// B) create a semaphore set with identifier key, if it doesn't already exist
semid = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);

// C) create a semaphore set with identifier key, but fail if it exists already
semid = semget(key, 10, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

La system call ***semctl*** esegue una varietà di operazioni di controllo su tutto l'insieme di semafori o su un solo semaforo all'interno dell'insieme.

Il codice identificativo:

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

// Returns nonnegative integer on success, or -1 error
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */);
```

Gli argomenti sono:

- ***semid***, l'identificatore dell'insieme del semaforo nel quale deve essere eseguito il controllo. Questo identificatore si ottiene tramite la syscall *semget*, presentata precedentemente.
- ***senum***, spiegata di seguito, identifica un semaforo specifico.
- ***cmd***, acronimo di *certain control operations*, richiede alcuni argomenti visti di seguito.

La unione *union* della syscall *semun* deve essere **esplicitamente definita dal programmatore** prima della chiamata *semctl*.

La definizione è la seguente:

```
#ifndef SEMUN_H
#define SEMUN_H
#include <sys/sem.h>
// definition of the union semun
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
#endif
```

Si ricorda che la *union* è come una *struct*, ma a livello di sistema operativo viene allocata in memoria in uno spazio grande quanto la variabile più grande al suo interno. Quindi, si può utilizzare una solo variabile per volta.

3.2 – Approfondimento utilizzo con FLAG (semctl)

Nello specifico si elencano i vari *flag*.

Il codice identificativo:

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

Si noti che *semnum* viene utilizzato il valore 0 per ignorare un semaforo specifico e quindi per riferirsi a tutto l'insieme.

I vari *flag* da utilizzare al posto di *cmd*:

- IPC_RMID: rimuove immediatamente l'insieme dei semafori e ogni processo relativo a quel semaforo che era in stato di blocco, viene risvegliato. In questo caso, l'argomento *arg* non deve essere usato.
- IPC_STAT: inserisce una copia della struttura *semi_ds* associata all'insieme dei semafori specificati, nel buffer di *arg*. Quindi, è necessario utilizzare *arg.buf*.
- IPC_SET: aggiorna i campi selezionati dalla struttura dati *semid_ds* associata all'insieme dei semafori. Attenzione! L'insieme di semafori a cui fa riferimento è puntato dal buffer *arg.buf*

In dettaglio, la struttura *semid_ds*:

```
struct semid_ds
{
    struct ipc_perm sem_perm; /* Ownership and permissions */
    time_t sem_otime;         /* Time of last semop() */
    time_t sem_ctime;         /* Time of last change */
    unsigned long sem_nsems;  /* Number of semaphores in set */
};
```

I campi sono:

- *sem_perm*, consente di cambiare i permessi e di vedere il controllore, la sua spiegazione è stata fatta ad inizio capitolo;
- *sem_otime*, consente di vedere il tempo dell'ultima syscall *semop*( );
- *sem_ctime*, consente di vedere il tempo dell'ultima modifica effettuata;
- *sem_nsems*, consente di vedere il numero di semafori in un insieme.

Solo i sottocampi *uid*, *gid* e *mode* della sottostruttura *sem_perm* può aggiornare i dati attraverso la *flag* IPC_SET.

Qui di seguito viene lasciato un pezzo di codice su **come cambiare i permessi** di un insieme di semafori:

```
ket_t key = //... (generate a key in some way, i.e. with ftok)

// get, or create, the semaphore set
int semid = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);

// instantiate a semid_ds struct
struct semid_ds ds;

// instantiate a semun union (defined manually somewhere)
union semun arg;
arg.buf = &ds;

// get a copy of semid_ds structure belonging to the kernel
if (semctl(semid, 0 /* ignored */, IPC_STAT, arg) == -1)
    errExit("semctl IPC_STAT failed");

// update permissions to guarantee read access to the group
arg.buf -> sem_perms.mode |= S_IRGRP;

// update the semid_ds structure of the kernel
if (semctl(semid, 0 /* ignored */, IPC_SET, arg) == -1)
    errExit("semctl IPC_SET failed");
```

Mentre per **rimuovere un insieme di semafori**:

```
if(semctl(semid, 0 /* ignored */, IPC_RMID, 0 /* ignored */) == -1)
    errExit("semctl failed");
else
    printf("semaphore set removed successfully\n");
```

Per **ottenere** il **valore** o **inizializzare** un **semaforo specifico**, si utilizza sempre la syscall *semctl*, ma con due flag:

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

Con il flag SETVAL il valore del *semnum*-esimo semaforo nell'insieme specificato da *semid*, è **inizializzato** al valore specifico in $arg.val$.

Con il flag GETVAL viene **ritornato il valore** del *semnum*-esimo semaforo nell'insieme specificato da *semid*. **Non** è richiesto l'argomento arg.

Per **ottenere** il **valore** o **inizializzare tutti** i **semafori**, si utilizza la syscall *semctl* con i seguenti flag:

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

Con il flag SETALL vengono **inizializzati** tutti i semafori dell'insieme riferito a *semid*, usando il valore fornito nell'array puntato da $arg.array$.

Con il flag GETALL si **ottengono i valori** di tutti i semafori. Essi vengono salvati nell'array puntato da $arg.array$.

Un esempio in cui si **inizializza un semaforo specifico** in un insieme di semafori:

```
ket_t key = //... (generate a key in some way, i.e. with ftok)

// get, or create, the semaphore set
int semid = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);

// set the semaphore value to 0
union semun arg;
arg.val = 0;

// initialize the 5-th semaphore to 0
if (semctl(semid, 5, SETVAL, arg) == -1)
    errExit("semctl SETVAL");
```

Un insieme di semafori deve essere sempre inizializzato prima di essere usato!

Un esempio in cui si **ottiene lo stato corrente** di un semaforo specifico:

```
ket_t key = //... (generate a key in some way, i.e. with ftok)

// get, or create, the semaphore set
int semid = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);

// get the current state of the 5-th semaphore
int value = semctl(semid, 5, GETVAL, 0 /* ignored */);
if (value == -1)
    errExit("semctl GETVAL");
```

Una volta ritornato il valore, il semaforo potrebbe avere il valore già cambiato!

Un esempio per mostrare come **inizializzare un insieme di semafori**:

```
ket_t key = //... (generate a key in some way, i.e. with ftok)

// get, or create, the semaphore 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");
```

Un insieme di semafori deve essere sempre inizializzato prima di essere utilizzato!

Con i seguenti flag si ottengono alcune informazioni:

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

Con il flag GETPID viene ritornato l'ID del processo che ha eseguito una operazione di tipo *semop*, ovvero che ha modificato il *semnum*-esimo semaforo.

Con il flag GETNCNT viene ritornato il numero di processi attualmente in attesa che un *semnum*-esimo semaforo diventi positivo.

Con il flag GETZCNT viene ritornato il valore dei processi attualmente in attesa che stanno aspettando che il valore del *semnum*-esimo semaforo diventi zero.

Un esempio di come si **ottengono informazioni su un semaforo** di un insieme di semafori:

```
ket_t key = //... (generate a key in some way, i.e. with ftok)

// get, or create, the semaphore set
int semid = semget(key, 10, IPC_CREAT | S_IRUSR | S_IWUSR);

// ...
// get information about the first semaphore of the semaphore set
printf("Sem:%d getpid:%d getncnt:%d getzcnt:%d\n",
semid,
semctl(semid, 0, GETPID, NULL),
semctl(semid, 0, GETNCNT, NULL),
semctl(semid, 0, GETZCNT, NULL));
```

Anche qui, **il valore ritornato potrebbe essere diverso da quello attuale a causa dei cambiamenti**.

3.3 – Altre operazioni

La system call *semop* può eseguire una o più operazioni ($wait(P)$ e $signal(V)$) sui semafori.

Il codice identificativo:

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

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

L'argomento *sops* è un puntatore ad un array contenente una sequenza ordinata di operazioni da eseguire atomicamente.

L'argomento *nsops* (> 0) fornisce la dimensione dell'array.

Gli elementi dell'array *sops* sono strutture nella seguente forma:

```c
struct sembuf
{
    unsigned short sem_num;  /* Semaphore number */
    short sem_op;            /* Operation to be performed */
    short sem_flg;           /* Operation flags */
};
```

È una **struttura che deve definirla il programmatore!**

Il campo *sem_num* identifica il semaforo, all'interno dell'insieme, in cui l'operazione deve essere eseguita.

Il campo *sem_op* specifica l'operazione da eseguire:

- $sem\_op > 0$, il valore di *sem_op* è aggiunto al valore del *sem_num*-esimo semaforo.
- $sem\_op = 0$, il valore del *sem_num*-esimo semaforo è controllato per vedere se è attualmente uguale a zero. Se non lo è, il processo chiamante viene bloccato finché il semaforo non è uguale a zero.
- $sem\_op < 0$, il valore del *sem_num*-esimo semaforo viene diminuito con un valore pari a quello specificato nel campo *sem_op*. In questo caso viene blocco il processo chiamante finché il valore del semaforo non è ad un livello tale che permetta le operazioni senza andare sottozero.

La syscall *semop* può bloccarsi in alcuni casi e il processo rimane bloccato finché:

- Un altro processo modifica il valore del semaforo tale che l'operazione richiesta può procedere;
- Un segnale *interrupt* interrompe la chiamata *semop*( ). In questo caso risulterà l'errore EINTR;
- Un altro processo elimina il semaforo riferito al *semid* del processo bloccato. In questo caso, la chiamata *semop* fallirà con l'errore EIDRM.

È possibile prevenire il blocco della system call *semop* inserendo il flag IPC_NOWAIT nel corrispondente campo *sem_flg* durante l'esecuzione di un'operazione su un particolare semaforo. In questo caso, se la syscall si dovesse bloccare, fallirebbe subito con l'errore EAGAIN.

Un esempio su **come inizializzare un array** di operazioni *sembuf*:

```
struct sembuf sops[3];

sops[0].sem_num = 0;
sops[0].sem_op = -1 // subtract 1 from semaphore 0
sops[0].sem_flg = 0;

sops[1].sem_num = 1;
sops[1].sem_op = 2; // add 2 to semaphore 1
sops[1].sem_flg = 0;

sops[2].sem_num = 2;
sops[2].sem_op = 0; // wait for semaphore 2 to equal 0
sops[2].sem_flg = IPC_NOWAIT; // but don't block if operation cannot be performed
immediately
```

Un esempio di come **eseguire operazioni su un insieme di semafori**:

```
struct sembuf sops[3];

// .. see the previous code to initilize sembuf

if (semop(semid, sops, 3) == -1) {
    if (errno == EAGAIN) // Semaphore 2 would have blocked
        printf("Operation would have blocked\n");
    else
        errExit("semop"); // Some other error
}
```

4.1 – Concetti fondamentali dei segnali

Un **segnale** è una notifica ad un processo che si è verificato un evento. Essi interrompono il normale flusso di esecuzione di un programma e in molti casi, non è possibile prevedere esattamente quando arriverà un segnale.

Si dice che un segnale viene **generato da qualche evento**. Una volta generato, viene successivamente consegnato ad un processo. Nel tempo in cui è stato generato il segnale e consegnato, il segnale viene detto in **stato di attesa** ($pending$).

Normalmente, un segnale viene consegnato ad un processo il prima possibile nel momento in cui acquista lo scheduler, ovvero va in esecuzione, altrimenti, nel caso in cui sia già in esecuzione, viene consegnato immediatamente.

Appena un segnale viene consegnato, il processo esegue una delle seguenti azioni a seconda del tipo di segnale:

- [$killed$] Il processo è terminato.
- [$stopped$] Il processo è sospeso.
- [$resumed$] Il processo è stato ripreso dopo una sospensione.
- [$ignored$] Il segnale viene ignorato. Viene scartato dal kernel e non ha effetti sul processo.
- [$signal\ handler$] Il processo esegue un segnale personalizzato, ovvero una funzione scritta appositamente dal programmatore che esegue compiti scritti *ad-hoc*.

4.2 – Tipi di segnali

Qui di seguito alcuni segnali per **terminare un processo**:

- **SIGTERM**, è segnale di terminazione che consente di eseguire alcune operazioni di prima della sua terminazione. Un codice ben strutturato dovrebbe avere sempre questo segnale per garantire un'uscita "elegante".
- **SIGINT**, termina un processo (*interrupt process*) ma consente di fare qualche operazione al processo che sta terminando. Questo segnale viene generato quando si utilizza la scorciatoia CTRL + C nel terminale.
- **SIGQUIT**, termina un processo e forza l'esecuzione di un *core dump*, il quale può essere usato per fare debug.
- **SIGKILL**, termina un processo in modo forzato! Non può essere bloccata, ignorata, o fermata in qualsiasi altro modo.

I segnali per **fermare** e **riprendere** un processo:

- **SIGSTOP**, ferma **sempre** un processo. È analogo al SIGKILL.
- **SIGCONT**, riprende a un processo precedentemente sospeso.

Altri segnali importanti:

- **SIGPIPE**, viene generato quando un processo prova a scrivere in una PIPE, in una FIFO per il quale non c'è il corrispondente processo lettore (vedere il capitolo sulle PIPE e FIFO).
- **SIGALRM**, viene inviato ad un processo dopo un tempo definito.
- **SIGUSR1** e **SIGUSR2** sono segnali personalizzabili per i programmatori. Il kernel non genererà **mai** questi tipi di segnali per un processo.

Scheda riassuntiva:

name	number	can be caught?	default action
SIGTERM	15	yes	terminates a process
SIGINT	2	yes	terminates a process
SIGQUIT	3	yes	dumps + terms a process
SIGKILL	9	no	terminates a process
SIGSTOP	17	no	stops a process
SIGCONT	19	yes	resumes a stopped process
SIGPIPE	13	yes	terminates a process
SIGALRM	14	yes	terminates a process
SIGUSR1	30	yes	terminates a process
SIGUSR2	31	yes	terminates a process

4.3 – Manipolazione di un segnale (signalhandler)

Un segnale manipolato (*signal handler*) è una funzione che viene invocata quando un segnale specifico è consegnato ad un processo.

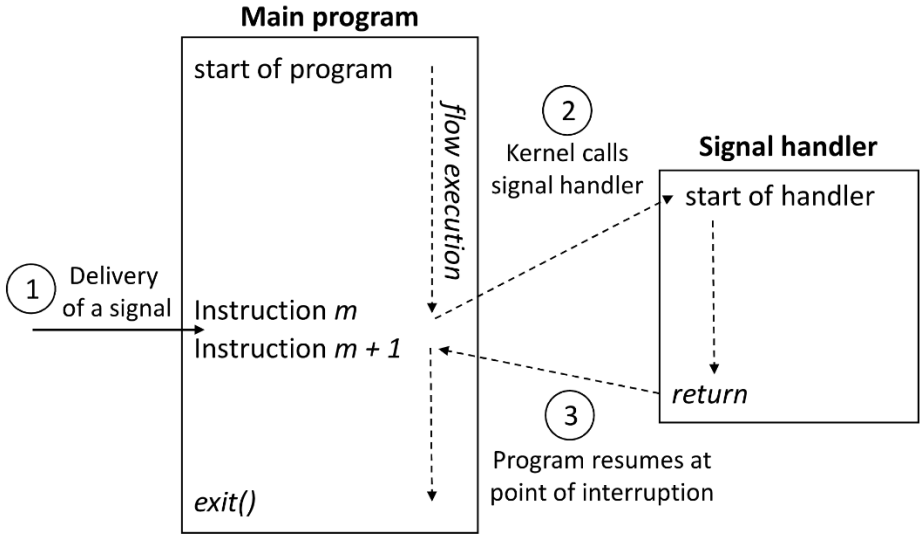
La sua forma generale:

```
void sigHandler(int sig) {
    /* Code for the handler */
}
```

Questa funzione non ritorna nessun valore e prende come argomenti un valore intero chiamato *sig*. Quando questo segnale viene invocato dal kernel, *sig* viene impostato al numero di segnale consegnato al processo.

Tipicamente, *sig* viene usato per "catturare" alcuni segnali e fare delle operazioni aggiuntive.

Quando viene invocata la funzione di manipolazione, il flusso d'esecuzione del programma *main* si interrompe. Esso riprende solamente quando la funzione ha terminato le sue istruzioni:



La **chiamata di sistema** *signal* consente di comunicare al kernel la manipolazione del segnale e quindi il codice è:

```c
#include <signal.h>

typedef void (*signalhandler_t) (int);
// Returns previous signal disposition on success, or SIG_ERR on error
signalhandler_t signal(int signum, signalhandler_t handler);
```

In cui il parametro *signum* identifica il segnale messo a disposizione che si vorrebbe gestire (il nome vero e proprio), mentre il parametro *handler* può essere specificata come:

- L'indirizzo di un segnale manipolato e definito dall'utente, ovvero quella della pagina precedente;
- La costante SIG_DFL, la quale resetta il valore di disposizione di default del processo per il segnale *signum*;
- La costante SIG_IGN, la quale imposta il processo per ignorare la consegna del segnale *signum*, ovvero ignora il segnale.

Un esempio:

```c
void sigHandler(int sig) {
    printf("The signal %s was caught!\n",
        (sig == SIGINT)? "Ctrl-C" : "signal User-1");
}

int main (int argc, char *argv[]) {
    // setting sigHandler to be executed for SIGINT or SIGUSR1
    if (signal(SIGINT, sigHandler) == SIG_ERR ||
        signal(SIGUSR1, sigHandler) == SIG_ERR) {
            errExit("change signal handler failed");
        }

    // Do something else here. During this time, if SIGINT/SIGUSR1
    // is delivered, sigHandler will be used to handle the signal.
    // Reset the default process disposition for SIGINT and SIGUSR1
    if (signal(SIGINT, SIG_DFL) == SIG_ERR ||
        signal(SIGUSR1, SIG_DFL) == SIG_ERR) {
            errExit("reset signal handler failed");
        }

    return 0;
}
```

Per **riassumere**:

- SIGKILL e SIGSTOP non possono essere gestite, *brute force*.
- Un segnale è un evento asincrono. Non è possibile predeterminare quando arriverà.
- Quando una la funzione di una manipolazione di un segnale (*signal handler*) è invocata, il segnale che causa la sua invocazione viene bloccato e successivamente sbloccato solo quando sarà finita l'esecuzione della funzione invocata.
- Un segnale bloccante può essere generato molteplici volte, ma basterà un semplice segnale per riprendere l'esecuzione.
- L'esecuzione di una manipolazione di segnale può essere interrotta dalla consegna di un segnale di sbloccaggio.
- La gestione dei segnali viene tramandata da padre a figlio.

La system call *pause* **sospende** l'esecuzione del processo finché non viene ricevuto un altro segnale:

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

La **funzione** *sleep* **addormenta** l'esecuzione di un processo per un numero di secondi specificati come argomento o come segnato manipolato:

```c
#include <unistd.h>
// Always return -1 with errno set to EINTR
unsigned int sleep(unsigned int seconds);
// Returns 0 on normal completion, or number of unslept seconds if prematurely
terminated
```

Un esempio di sospensione per 30 secondi e una verifica continua della ricezione del segnale CTRL + C:

```c
void sigHandler(int sig) {
    printf("Well done!\n");
}

int main (int argc, char *argv[]) {
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        errExit("change signal handler failed");

    int time = 30;
    printf("We can wait for %d seconds!\n", time);
    time = sleep(time); // the process is suspended for max. 30sec.
    printf("%s\n", (time == 0) ? "out of time", "just in time");
}
```

4.4 – Invio dei segnali

La system call *kill* consente a un processo di inviare un segnale ad un altro processo:

```c
#include <signal.h>

// Returns 0 on success, or -1 on error
int kill(pid_t pid, int sig);
```

L'argomento *pid* identifica uno o più processi ai quali il segnale *sig* deve essere inviato. Se il valore di *pid* è:

- $pid > 0$, il segnale è inviato al processo avente come PID il valore nell'argomento *pid*;
- $pid = 0$, il segnale è inviato ad ogni processo appartenente al gruppo del processo chiamante, incluso il processo chiamante stesso;
- $pid < 0$, il segnale è inviato a tutti i processi nel gruppo di processi il cui ID è uguale al valore assoluto dell'argomento *pid*;
- $pid = -1$, il segnale è inviato a tutti i processi per cui il processo chiamante ha il permesso di inviare il segnale, eccetto il processo *init* e sé stesso.

Un esempio di codice che **invia** un **segnale** SIGKILL ad un **processo figlio**:

```c
int main (int argc, char *argv[]) {
    pid_t child = fork();
    switch (child)
    {
    case -1:
        errExit("fork");

    case 0:     /* Child process */
        while(1);   // <- child is stuck here!

    default:    /* Parent process */
        sleep(10);              // wait 10 seconds
        kill(child, SIGKILL);   // kille the child process
    }

    return 0;
}
```

La system call *alarm* consente di **inviare** il **segnale** SIGALRM al **processo chiamante** dopo un ritardo stabilito come parametro (*seconds*):

```c
#include <signal.h>

// Always succeeds, returning number of seconds remaining on
// any previously set timer, or 0 if no timer previously was set
unsigned int alarm(unsigned int seconds);
```

Come detto in precedenza, l'argomento *seconds* specifica il numero di secondi da attendere per inviare il segnale.

Un esempio di utilizzo di *alarm* per verificare quanto tempo ha impiegato un pezzo di codice:

```c
void sigHandler(int sig) {
    printf("Out of time!\n");
    _exit(0);
}

int main(int argc, char *argv[]) {
    if (signal(SIGALRM, sigHandler) == SIG_ERR)
        errExit("change signal handler failed");

    int time = 30;
    printf("We have %d seconds to complete the job!\n", time);
    alarm(time); // setting a timer

    /* Do something else here. */

    time = alarm(0); // disabling timer, overwrite
    printf("%s seconds before timer expirations!\n", time);
    return 0;
}
```

4.5 – Impostare e bloccare un segnale

Il tipo di dato *sigset_t* rappresenta un insieme di segnali. La funzione *sigemptyset* e *sigfillset* vengono usate per inizializzare un insieme di segnali, prima di utilizzarli.

```c
#include <signal.h>

typedef unsigned long sigset_t;

// Both return 0 on success, or -1 on error.
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
```

La funzione *sigemptyset* inizializza un insieme di segnali senza nessun segnale di preciso al suo interno, mentre la funzione *sigfillset* crea un insieme contenente **tutti** i segnali del sistema operativo tranne SIGKILL e SIGSTOP.

Dopo l'inizializzazione, ogni **segnale** può essere **aggiunto** all'insieme con la funzione *sigaddset* e analogamente possono essere **rimossi** precisamente tramite *sigdelset*:

```c
#include <signal.h>

// Both return 0 on success, or -1 on error.
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

Per entrambe le funzioni, il parametro *sig* è il valore del segnale.

La funzione *sigismember* viene utilizzata per verificare la presenza di uno specifico segnale all'interno di un insieme:

```c
#include <signal.h>

// Returns 1 if sig is a member of set, otherwise 0
int sigismember(const sigset_t *set, int sig);
```

Per ogni processo, il kernel mantiene una maschera chiamata *signal mask*, ovvero un insieme di segnali la cui consegna ai processi è attualmente bloccata. Nel momento in cui un segnale bloccato viene inviato al processo, l'operazione di consegna viene **ritardata** (*delay*) finché non viene rimosso il segnale dalla maschera.

La chiamata di sistema *sigprocmask* può essere utilizzata in ogni momento per **aggiungere** e **rimuovere segnali** dalla *signal mask*:

```c
#include <signal.h>

// Returns 0 on success, or -1 on error
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

L'argomento *how* può essere uno dei tre *flag* seguenti, mentre il puntatore *set* punta all'insieme che si vuole impostare, eventualmente anche l'insieme vecchio:

- SIG_BLOCK, l'insieme di segnali bloccati, ovvero l'unione dell'insieme corrente e dell'insieme specificato come argomento (*set*);
- SIG_UNBLOCK, i segnali nell'argomento *set* sono rimossi dalla maschera, ovvero dall'insieme dei segnali bloccati. Quindi quei segnali possono raggiungere il processo;
- SIG_SETMASK, l'insieme di segnali bloccati è l'insieme dell'argomento *set*, quindi si definisce una nuova maschera.

In ogni caso, se l'argomento *oldset* non è NULL, esso punterà ad un buffer di tipo *sigset_t* che viene utilizzato per ritornare la precedente *signal mask*. Per ignorare la vecchia maschera, basta inserire NULL come argomento di *oldset*.

Un esempio di **blocco di tutti i segnali eccetto uno**, ovvero SIGTERM:

```c
int main (int argc, char *argv[])
{
    sigset_t mySet, prevSet;

    // initialize mySet to contain all signals
    sigfillset(&mySet);

    // remove SIGTERM from mySet
    sigdelset(&mySet, SIGTERM);

    // blocking all signals but SIGTERM
    sigprocmask(SIG_SETMASK, &mySet, &prevSet);

    // the process is not interrupted by signals except SIGTERM

    // reset the signal mask of the process
    sigprocmask(SIG_SETMASK, &prevSet, NULL);
    // the process is not interrupted by signals in prevSet
    return 0;
}
```

Capitolo 5 – Memoria condivisa e coda di messaggi

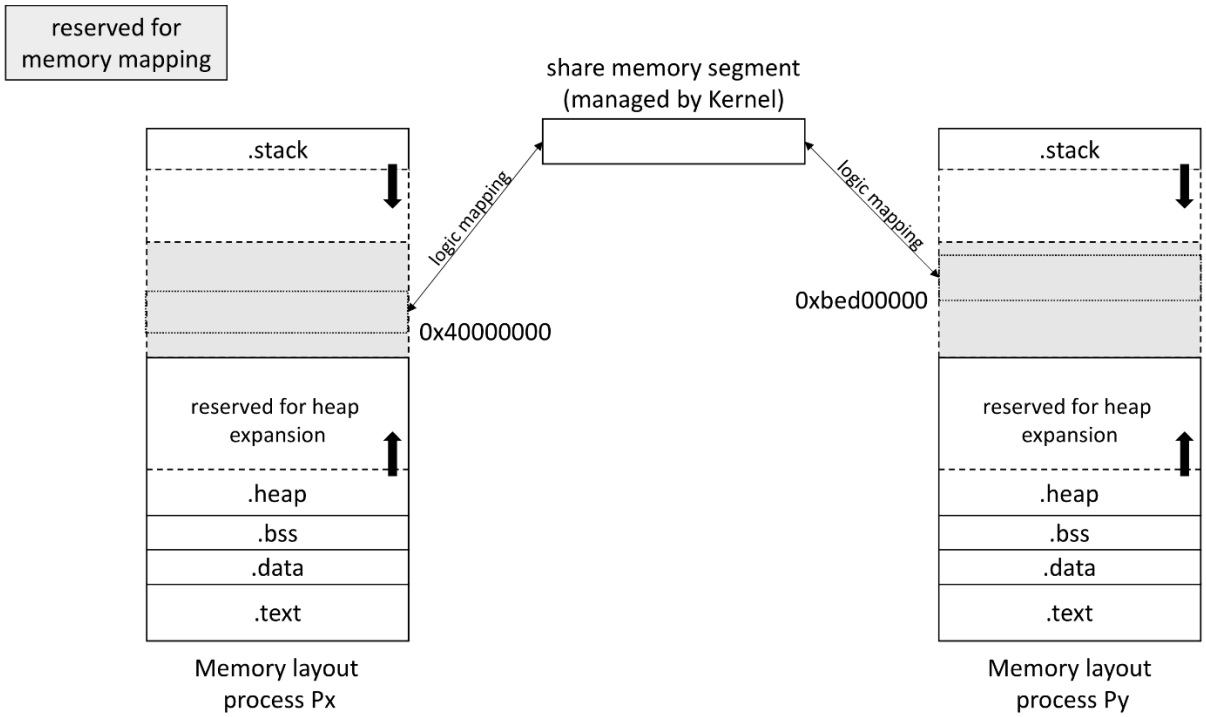
In questo capitolo vengono introdotte la memoria condivisa (*shared memory*) e la coda di messaggi (*message queue*).

1 – Memoria condivisa (*shared memory*)

Una memoria condivisa è un **segmento di memoria fisica** gestita dal Kernel, in cui due o più **processi possono scambiarsi dati**.

Una volta eseguita l'operazione di memoria condivisa su un processo, essa sarà parte dello spazio di indirizzo virtuale e l'intervento del kernel non sarà più richiesto.

I dati scritti in una memoria condivisa sono immediatamente disponibili a tutti gli altri processi condivisi nello stesso segmento di memoria. Tipicamente, alcuni metodi di sincronizzazione sono richiesti per gestire l'accesso a tale area, per esempio i semafori.



1.1 – Creazione (shmget)

La syscall *shmget* **crea** un nuovo segmento di **memoria condivisa** oppure **ottiene** l'identificatore di una già esistente. Il contenuto di una nuova memoria condivisa appena creata è sempre inizializzato in automatico a 0:

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

// Returns a shared memory segment identifier on success, or -1 on error
int shmget(key_t key, size_t size, int shmflg);
```

Gli argomenti sono:

- *key*, una chiave IPC;
- *size*, la dimensione desiderata, in bytes, del segmento condiviso. Nel caso in cui si voglia ottenere un segmento esistente, questo parametro non ha effetto, ma **deve** essere minore o uguale alla dimensione del segmento ricercato;
- *shmflg*, è una *bit mask* che specifica i permessi (come nella *open*( )) da utilizzare in una nuova memoria condivisa o in una già esistente. In aggiunta alle FLAG della syscall *open*, ci sono anche:
  - IPC_CREAT, se non esistono segmenti con la chiave specificata (*key*), viene creato un segmento;
  - IPC_EXCL, usata con IPC_CREAT, fa fallire la syscall stessa (*shmget*) se esiste un segmento con la chiave specificata (*key*).

Un **esempio** di come **creare una memoria condivisa**:

```c
int shmid;
ket_t key =   //... (generate a key in some way, i.e. with fork)
size_t size = //... (compute size value in some way)

// A) delegate the problem of finding a unique key to the kernel
shmid = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

// B) create a shared memory with identifier key, if it doesn't already exist
shmid = shmget(key, size, IPC_CREAT | S_IRUSR | S_IWUSR);

// C) create a shared memory with identifier key, but fail if it exists already
shmid = shmget(key, size, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

1.2 – Assegnazione (shmat)

La syscall *shmat* consente di assegnare la memoria condivisa, identificata da *shmid*, al processo chiamante:

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

// Returns address at which shared memory is attached on success
// or (void *)-1 on error
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Gli argomenti sono:

- *shmid*, identifica la memoria condivisa da assegnare;
- *shmaddr* può essere:
  - NULL, il segmento viene assegnato dal kernel a qualche indirizzo;
  - diverso da NULL, il segmento viene inserito all'indirizzo specificato in *shmaddr*;
- *shmflg*:
  - SHM_RND, viene posizionato all'indirizzo più vicino alla costante SHMLBA;
  - SHM_RDONLY, nella memoria condivisa l'accesso sarà solo in lettura

Solitamente il parametro *shmaddr* viene messo a NULL poiché aumenta la portabilità dell'applicazione, dato che non dipenderà solo da quella specifica architettura, e inoltre è rischioso scrivere un indirizzo perché si aumenta la probabilità di sovrascrivere un'area di memoria già occupata.

Un esempio di come **assegnare una memoria condivisa**:

```c
// 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]);
```

1.3 – De-assegnazione (shmdt)

La syscall *shmdt* consente di de-assegnare la memoria condivisa al processo chiamante:

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

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

In questo caso, l'argomento *shmaddr* identifica il segmento da de-assegnare e tale valore è ottenibile con la syscall precedente (*shmat*).

Durante una *execution*, tutte le assegnazioni alla memoria condivisa vengono de-assegnate automaticamente. Anche quando un processo termina la sua assegnazione viene persa.

Un **esempio** di come **de-assegnare una memoria condivisa**:

```c
// 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)
    errExit("shmdt failed");
```

1.4 – Operazioni di controllo (shmctl)

La syscall *shmdt* consente di de-assegnare la memoria condivisa al processo chiamante:

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

// Returns 0 on success, or -1 on error
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

L'argomento *shmid* è un identificatore della memoria condivisa.

L'argomento *cmd* specifica l'operazione da eseguire nella memoria condivisa:

- IPC_RMID, la memoria condivisa viene contrassegnata come "da eliminare". Il segmento sarà rimosso il prima possibile appena tutti i processi avranno fatto la de-assegnazione;
- IPC_STAT, inserimento di una copia di *shmid_ds*, la struttura dati della memoria condivisa selezionata, nel buffer puntato da *buf*;
- IPC_SET, aggiornamento dei campi della struttura dati *shmid_ds* associata alla memoria condivisa con i campi all'interno del registro puntato da *buf*.

Un **esempio di come rimuovere una memoria condivisa**:

```
if (shmctl(shmid, IPC_RMID, NULL) == -1)
    errExit("shmctl failed");
else
    printf("shared memory segment removed successfully\n");
```

Per ogni memoria condivisa, il kernel mantiene la seguente struttura dati:

```
struct shmid_ds
{
    struct ipc_perm shm_perm;  /* Ownership and permissions     */
    size_t shm_segsz;          /* Size of segment in bytes      */
    time_t shm_atime;          /* Time of last shmat()          */
    time_t shm_dtime;          /* Time of last shmdt()          */
    time_t shm_ctime;          /* Time of last change           */
    pid_t shm_cpid;            /* PID of creator                */
    pid_t shm_lpid;            /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch;       /* Number of currently attached  */
};                             /* processes                     */
```

Con i flags IPC_STAT e IPC_SET è possibile rispettivamente ottenere e aggiornare questa struttura dati. L'unico campo modificabile è *shm_perm*.

2.1 – Creazione (msgget)

La syscall *msgget* crea una nuova coda di messaggi oppure ottiene l'identificatore nel caso in cui esista già:

```c
#include <sys/msg.h>

// Returns message queue identifier on success, or -1 error
int msgget(key_t key, int msgflg);
```

L'argomento *key* è una chiave IPC, mentre *msgflg* è una *bit mask* che specifica i permessi (permessi usati da *open*( ), l'argomento *mode*). In più ci sono altri flags:

- IPC_CREAT, se non ci sono code di messaggi con una chiave esistente, viene creata una nuova coda;
- IPC_EXCL, insieme a IPC_CREAT, fa fallire la syscall *msgget* se una coda esiste con la chiave *key*.

Un **esempio di come creare una coda di messaggi**:

```c
int msqid;
ket_t key = //... (generate a key in some way, i.e. with ftok)

// A) delegate the problem of finding a unique key to the kernel
msqid = msgget(IPC_PRIVATE, S_IRUSR | S_IWUSR);

// B) create a queue with identifier key, if it doesn't already exist
msqid = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);

// C) create a queue with identifier key, but fail if it exists already
msqid = msgget(key, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

2.2 – Struttura di un messaggio e invio (msgsnd)

Un messaggio in una coda di messaggi è sempre strutturato come segue:

```
struct mymsg
{
    long mytype;  // Message type
    char mtext[]; // Message body
};
```

La prima parte di un messaggio contiene il tipo di messaggio, specificato da una variabile *long* maggiore di 0. Il campo *mtext* può essere omesso.

La syscall *msgsnd* consente di scrivere un messaggio:

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

// Returns 0 on success, or -1 on error
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

Gli argomenti sono:

- *msqid*, è l'ID della coda dei messaggi su cui scrivere (ottenuta con *get*);
- *msgp*, è un puntatore alla struttura del messaggio creato;
- *msgsz*, il numero di byte contenuti nel campo *mtext* del messaggio;
- *msgflg*, può essere 0 oppure il flag IPC_NOWAIT:
    - Con 0, se una coda di messaggi è piena, *msgsnd* si blocca finché non viene letto un messaggio;
    - Se è specificato IPC_NOWAIT, la *msgsnd* ritorna immediatamente un errore di tipo EAGAIN (*i. e.* non ci sono dati disponibili adesso, riprovare più avanti).

Un **esempio**:

```
// Message structure
struct mymsg
{
    long mtype;
    char mtext[100] // Array of chars as message body
} m;

// message has type 1
m.mtype = 1;

// message contains the following string
char *text = "Ciao mondo!";
memcpy(m.mtext, text, strlen(text) + 1) // why +1 here?
// size of m is only the size of its mtext attribute!

size_t mSize = sizeof(struct mymsg) - sizeof(long);

// sending the message in the queue
if (msgsnd(msqid, &m, mSize, 0) == -1)
    errExit("msgsnd failed");
```

Un altro **esempio**:

```
// Message structure
struct mymsg
{
    long mtype;
    int num1, num2; // two integers as message body
} m;

// message has type 2
m.mtype = 2;

// message contains the following numbers
m.num1 = 34;
m.num2 = 43;

// size of m is only the size of its mtext attribute!
size_t mSize = sizeof(struct mymsg) - sizeof(long);

// sending the message in the queue
if (msgsnd(msqid, &m, mSize, 0) == -1)
    errExit("msgsnd failed");
```

Un altro **esempio**:

```c
// Message structure
struct mymsg
{
    long mtype;
    // The message has not got body. It has just a type!
} m;

// message has type 3
m.mtype = 3;

// size of m is only the size of its mtext attribute!
size_t mSize = sizeof(struct mymsg) - sizeof(long); // 0!

// sending the message in the queue
if (msgsnd(msqid, &m, mSize, 0) == -1) {
    if (errno == EAGAIN) {
        printf("The queue was full!\n");
    } else {
        errExit("msgsnd failed");
    }
}
```

2.3 – Ricezione di un messaggio (msgrcv)

La syscall *msgrcv* legge e **rimuove** un messaggio da una coda di messaggi:

```c
#include <sys/msg.h>

// Returns number of bytes copied into msgp on success, or -1 error
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg);
```

L'argomento *msqid* è l'identificatore della coda dei messaggi.

L'argomento *msgp* è il puntatore ad una struttura dove viene salvato il messaggio.

L'argomento *msgsz* è la dimensione del messaggio in byte.

L'argomento *msgtype* può essere usato per estrarre un messaggio specifico.

Infine, l'argomento *msgflg* sono i flags.

Il valore nel campo *msgtype* seleziona il messaggio recuperato nel seguente modo:

- Se uguale a 0, vengono letti tutti i messaggi all'interno della coda;
- Se maggiore di 0, viene preso il primo messaggio con quel valore;
- Se minore di 0, viene preso il valore assoluto, vengono estratti i valori dalla coda che hanno un *mytipe* minore o uguale al valore assoluto.

Per esempio:

> And the following queue:
>
> {(300,'a'); (100,'b'); (200,'c'); (400,'d'); (100,'e')}
>
> A series of `msgrcv` calls with `msgtype=-300` retrieve the messages:
>
> (100,'b'), (100,'e'), (200,'c'), (300,'a')

I flag *msgflg* è un *bit mask* formato da OR:

- IPC_NOWAIT, se non viene trovato il messaggio specifico, ritorna l'errore ENOMSG;
- MSG_NOERROR, se la dimensione del campo *mtext*, escluso il *mtype*, è più grande del *msgsz* fornito come argomento, ritorna l'errore ma anche il messaggio (è consigliato usare questo flag al posto di IPC_NOWAIT).

Un **esempio**:

```
// Message structure
struct mymsg
{
    long mtype;
    char mtext[100] // Array of chars as message body
} m;

// Get the size of the mtext field.
size_t mSize = sizeof(struct mymsg) - sizeof(long);

// Wait for a message having type equals to 1
if (msgrcv(msqid, &m, mSize, 1, 0) == -1)
    errExit("msgrcv failed");
```

Un altro esempio:

```
// Message structure
struct mymsg
{
    long mtype;
    char mtext[100] // Array of chars as message body
} m;

// Set an arbitrary size for the size.
size_t mSize = sizeof(char) * 50;

// Wait for a message having type equals to 1, but copy its first 50 bytes only
if (msgrcv(msqid, &m, mSize, 1, MSG_NOERROR) == -1)
    errExit("msgrcv failed");
```

Un altro esempio ancora:

```
// Message structure
struct mymsg
{
    long mtype;
} m;

// In polling mode, try to get a message every SEC seconds.
while (1)
{
    sleep(SEC);
    // Performing a nonblocking msgrcv.
    if (msgsnd(msqid, &m, 0, 3, IPC_NOWAIT) == -1)
    {
        if (errno == ENOMSG)
        {
            printf("No message with type 3 in the queue\n");
        }
        else
        {
            errExit("msgrcv failed");
        }
    }
    else
    {
        printf("I found a message with type 3\n");
    }
}
```

2.4 – Operazioni di controllo (msgctl)

La syscall *msgctl* esegue alcune operazioni di controllo sulla coda di messaggi:

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

// Returns 0 on success, or -1 error
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

L'argomento *msqid* è un identificatore della coda dei messaggi.

L'argomento *cmd* specifica l'operazione da eseguire sulla coda:

- IPC_RMID, viene rimossa l'intera coda di messaggi, tutti i messaggi non letti vengono persi e ogni processo bloccato in lettura/scrittura verrà svegliato e gli verrà ritornato *errno* impostato a EIDRM. In questa operazione *buf* è ignorato;
- IPC_STAT, viene inserita una copia della struttura *msqid_ds* della coda dei messaggi all'interno del registro puntato da *buf*;
- IPC_SET, aggiorna la struttura *msqid_ds* della coda dei messaggi con quella inserita nel registro puntato da *buf*.

Un **esempio di come rimuovere una coda di messaggi**:

```
if (msgctl(msqid, IPC_RMID, NULL) == -1)
    errExit("msgctl failed");
else
    printf("message queue removed successfully\n");
```

Per ogni coda di messaggi, il kernel ha associato una struttura dati chiamata *msqid_ds*:

```
struct msqid_ds
{
    struct ipc_perm msg_perm;   /* Ownership and permissions   */
    time_t msg_stime;           /* Time of last msgsnd()       */
    time_t msg_rtime;           /* TIme of last msgrcv()       */
    time_t msg_ctime;           /* Time of last change         */
    unsigned long __msg_cbytes; /* Number of bytes in queue    */
    msgqnum_t msg_qnum;         /* Number of messages in queue */
    msglen_t msg_qbytes;        /* Maximum bytes in queue      */
    pid_t msg_lspid;            /* PID of last msgsnd()        */
    pid_t msg_lrpid;            /* PID of last msgrcv()        */
};
```

Con IPC_STAT e IPC_SET è possibile rispettivamente prendere e aggiornare i dati della struttura.

Un **esempio di come cambiare limite di dimensione superiore di una coda di messaggi**:

```
struct msqid_ds;
// Get the data structure of a message queue
if (msgctl(msqid, IPC_STAT, &ds) == -1)
    errExit("msgctl");

// Change the upper limit on the number of bytes in the mtext
// fields of all messages in the message queue to 1 Kbyte
ds.msg_qbytes = 1024;

// Update associated data structure in kernel
if (msgctl(msqid, IPC_SET, &ds) == -1)
    errExit("msgctl");
```

3 – Riepilogo interfacce System V IPC

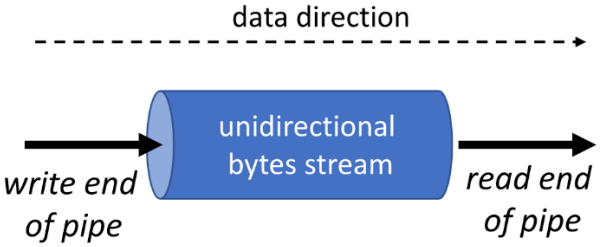
Interface	Message queues	Semaphores	Shared memory
Header file	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Data structure	`msqid_ds`	`semid_ds`	`shmid_ds`
Create/Open	`msgget(...)`	`semget(...)`	`shmget(...)`
Close	`(none)`	`(none)`	`shmdt(...)`
Control Oper.	`msgctl(...)`	`semctl(...)`	`shmctl(...)`
Performing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

Capitolo 6 – PIPE e FIFO

In questo capitolo vengono introdotti gli algoritmi di sostituzione delle pagine: PIPE e FIFO.

1 – PIPE

Una PIPE è una sequenza di byte (*byte stream*), la quale consente ai processi di scambiarsi bytes. Tecnicamente parlando, si può far riferimento alle PIPE come a un buffer posizionato nella memoria del kernel.



Una PIPE ha le seguenti **proprietà**:

- È **unidirezionale**. I dati viaggiano solo in una direzione, quindi vengono scritti i dati da un lato e analogamente vengono letti i dati dall'altro;
- I dati sono **sequenziali**. I bytes vengono letti dalla PIPE esattamente nell'ordine in cui sono stati scritti;
- Non esiste una concezione di **messaggi**. Il processo che esegue una lettura (*read*) da una PIPE, può leggere blocchi di qualsiasi dimensione indipendentemente dalla dimensione del blocco scritto dal processo scrittore. Quindi, un funzionamento con i messaggi può essere implementato, ma di base non esiste;
- Tentativi di lettura di un blocco vuoto bloccheranno il processo chiamante finché:
  - Non verrà effettuata una scrittura da qualche altro processo;
  - Non arriverà un segnale di errore con *errno* impostato a EINTR.
- Quando il lato di scrittura della PIPE viene chiuso, il rispettivo processo lettore vedrà un "*end-of-file*" indicante che la parte opposta (scrittura) è stata chiusa;
- Un processo in scrittura viene bloccato se:
  - Non c'è più spazio disponibile sulla PIPE (la dimensione di una PIPE su Linux è di 65′536 bytes);
  - Viene ricevuto un segnale di errore con *errno* impostato a EINTR;
- Eseguire un tentativo di scrittura superiore al buffer PIPE_BUF (su Linux ha dimensione 4′096 bytes) viene generato un errore dal sistema operativo.
- La scrittura è **atomica**, quindi non esistono conflitti nel momento in cui due processi, o più, scrivono un messaggio.

1.1 – Creazione e utilizzo delle PIPE

La system call *pipe* crea una nuova PIPE:

```c
#include <unistd.h>

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

Dato come argomento un array vuoto, in esso vengono salvati due *open file descriptors*:

- filedes[0] → *file descriptor* del lato del **lettore** della PIPE;
- filedes[1] → *file descriptor* del lato dello **scrittore** della PIPE.

Come ogni altro *file descriptor*, è possibile utilizzare le system call *read* e *write* per eseguire operazioni di I/O sulla PIPE.

[**Principale differenza con le FIFO**] Normalmente, la PIPE viene utilizzare per consentire la comunicazione tra processi che hanno una relazione tra di loro, ovvero rapporti di parentela (esempio padre e figlio).

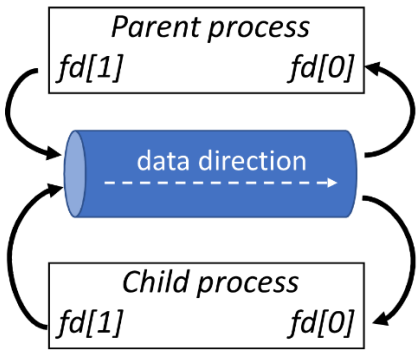
Per connettere due processi usando una PIPE, è necessario eseguire la syscall *fork*( ) subito **dopo** la *pipe*( ).

Esempio di **creazione PIPE**:

```
int fd[2];

//checking if PIPE successed
if (pipe(fd) == -1)
    errExit("PIPE");

// Create a child process
switch (fork())
{
    case -1:
        errExit("fork");

    case 0:
        // ...child reads from PIPE
        // see next code!
        break;

    default: // Parent
        // ...parent writes to PIPE
        // see next code!
        break;
}
```

La system call *pipe( )* crea una nuova *pipe* e salva all'interno dell'array *fd* i due *file descriptor* che si riferiscono alla lettura (indice 0) e scrittura (indice 1).

La system call *fork( )* crea un processo figlio, il quale eredita la tabella *file descriptor* del padre:



La lettura del figlio che si riferisce al codice precedente (*case* 0):

```
char buf[SIZE];
ssize_t nBys;

// close unused write-end
if (close(fd[1]) == -1)
    errExit("close - child");

// reading from the PIPE
nBys = read(fd[0], buf, SIZE);

// 0: end-of-file, -1: failure
if (nBys > 0) {
    buf[nBys] = '\0';
    printf("%s\n", buf);
}

// close read-end of PIPE
if (close(fd[0]) == -1)
    errExit("close - child");
```

La scrittura del padre che si riferisce al codice precedente (*case default*):

```
int fd[2];

//checking if PIPE successed
if (pipe(fd) == -1)
    errExit("PIPE");

// Create a child process
switch (fork())
{
    case -1:
        errExit("fork");

    case 0:
        // ...child reads from PIPE
        // see next code!
        break;

    default: // Parent
        // ...parent writes to PIPE
        // see next code!
        break;
}
```

2 – FIFO

Una FIFO è una sequenza di byte (*byte stream*), che consente ai processi di scambiarsi bytes. Come per le PIPE, anche le FIFO vengono salvate in un buffer nella memoria del kernel.

La **principale differenza** tra una PIPE e una FIFO, è che una FIFO consente la comunicazione tra processi che non hanno parentela. Inoltre, alla FIFO è possibile dare un nome, a differenza delle PIPE alle quali vengono forniti i *file descriptors*, al quale i processi senza parentela possono accedervi.

Anche la FIFO ha un lato di lettura e un lato di scrittura.

2.1 – Creazione, apertura e utilizzo

La system call *mkfifo* **crea una nuova FIFO**:

```c
#include <unistd.h>

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

Il parametro *pathname* specifica dove verrà creata la FIFO. Come un file normale, il parametro *mode* specifica i permessi per la FIFO (permessi identici a quelli della *open*).

Una volta che una FIFO è stata creata, ogni processo può aprirla.

La system call *open* consente di **aprire una FIFO**:

```c
#include <unistd.h>

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

Il parametro *pathname*, come per la *mkfifo*, specifica la locazione della FIFO nel file system. Il parametro *flags* è una *bit mask* che specifica gli accessi alla FIFO. Le costanti possono essere due:

- O_RDONLY, apertura della FIFO in sola lettura;
- O_WRONLY, apertura della FIFO in sola scrittura.

L'apertura di una FIFO in lettura (O_RDONLY) blocca il processo che sta eseguendo questa operazione finché la controparte, ovvero il processo in scrittura, non apre la FIFO (O_WRONLY).

Viceversa, un processo in scrittura viene bloccato finché un altro processo non apre la FIFO in lettura.

Quindi, la FIFO penserà alla **sincronizzazione** dei processi.

Un **esempio di un processo destinatario (lettore**, *reading*):

```c
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
unlink(fname);
```

Un **esempio di un processo mittente (scrittore**, *writing*):

```c
char *fname = "/tmp/myfifo";

// opening for writing only
int fd = open(fname, O_WRONLY);

// reading a str. (no spaces)
char buffer[LEN];
printf("Give me a string: ");
scanf("%s", buffer);

// writing the string on fifo
write(fd, buffer, strlen(buffer));

// closing the fifo
close(fd);
```