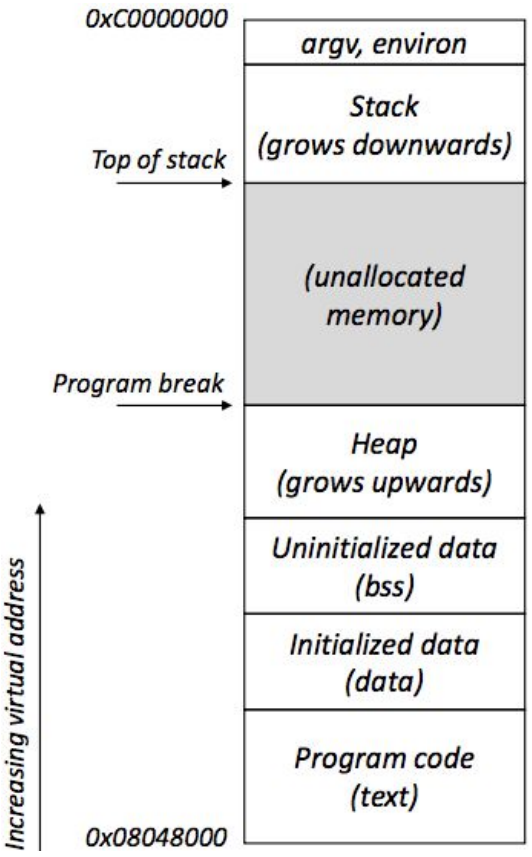
# SISTEMI OPERATIVI TEORIA LABORATORIO

## LEZIONE 0 - Concetti Fondamentali

**Programma**: file binario contenente una serie di informazioni necessarie per creare il processo
**Processo**: è un'istanza di un processo in esecuzione. Dal punto di vista del kernel, consiste in una zona di memoria user-space contenente il codice del programma, variabili, strutture dati del kernel per mantenere varie informazioni.

## Memory Layout di un processo

- **text**: segmento read-only, contiene le istruzioni in linguaggio macchina
- **initialized data (DATA)**: segmento che contiene le variabili globali e statiche inizializzate
- **uninitialized data (BSS)**: segmento che contiene le variabili globali e statiche non inizializzate
- **heap**: segmento che contiene le variabili allocate dinamicamente
- **stack**: segmento che contiene, per ogni funzione chiamata, i suoi argomenti e le variabili dichiarate localmente




Il comando `size` permette di vedere le dimensioni dei vari segmenti

**NOTA:** se dentro alla funzione main scrivo queste due istruzioni:

```
char *string = "ciao"
char string[] = "ciao"
```

La prima viene scritta in **text**, quindi se provo a modificarla mi da Segmentation Fault, la seconda nello **stack**

## File descriptor table

Per ogni processo il kernel mantiene una File Descriptor Table. Ogni elemento di questa tabella è una **file descriptor**, identificato da un numero positivo e che rappresenta una risorsa di input/output aperta dal processo.
Per convenzione, ci sono sempre 3 file descriptor in un nuovo processo:

- **1** standard input, STDIN_FILENO
- **2** standard output, STDOUT_FILENO
- **3** standard error, STDERR_FILENO

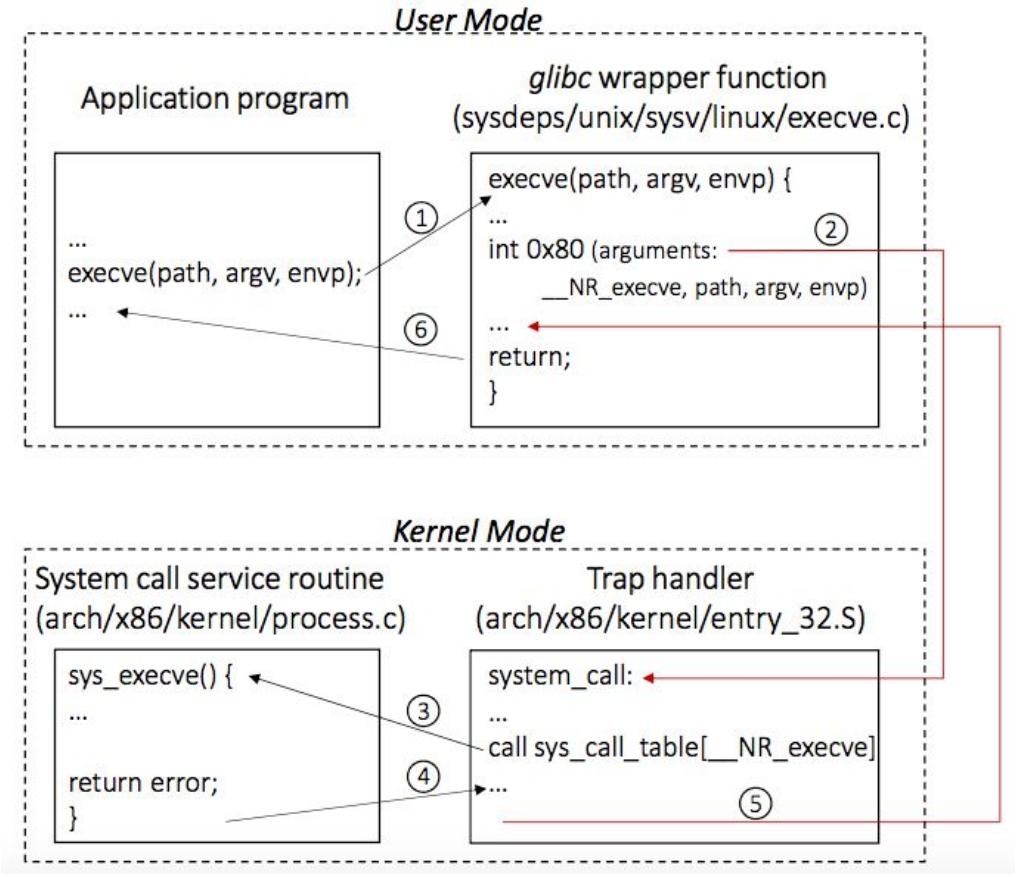
## System Call

Le system call sono un entry point controllato ai servizi messi a disposizione dal kernel. Servono perchè il processo lavora in user-space, che ha funzionalità limitate rispetto al kernel-space.

- L'applicazione esegue una system call chiamando una funzione wrapper della libreria C
- La funzione wrapper: copia gli argomenti della system call dallo stack in registri della cpu specifici, in particolare salva il **numero della system call** nel registro **%EAX**. Inoltre, esegue lo **switch mode da user-mode a kernel-mode**
- il **kernel** esegue la *system_call() routine* che salva i valori dei registri nello stack del kernel, controlla la validità del numero della system call, e invoca la system call service routine (la *sys_call_table* contiene i puntatori alle system call service routine)
- la service routine esegue l'attività richiesta e ritorna il *result status* alla *system_call*

- ➔ La *system_call() routine* ripristina i valori dei registri della cpu dal kernel stack e salva il **result status** nello stack. Simultaneamente, esegue lo **switch mode da kernel-mode a user-mode** e ritorna alla funzione wrapper.
- ➔ Se il valore di ritorno della system call service routine indica un errore, la funzione wrapper setta la variabile globale `errno`. Infine, la funzione wrapper ritorna al chiamante un valore intero che indica se ha avuto successo o meno

Per convenzione, il numero negativo -1 (o NULL) indica un errore, ma bisogna sempre verificare nella documentazione.




## Gestione degli errori

La sezione **ERRORS** nel manuale di ogni system call documenta i possibili valori di ritorno che indicano un errore.

Per poter utilizzare la variabile globale `errno` bisogna includere un header che fornisce la dichiarazione di questa variabile → `#include <errno.h>`

**Funzioni utili per il debug**

- ➔ `void perror (const char *msg)` : prepende un messaggio al messaggio di errore
- ➔ `char *strerror (int errnum)` : ritorna la stringa che descrive un error number e può essere sovrascritta

Il comando `strace` permette di vedere quali system call sta usando un processo.

## Kernel data types

Sono stati definiti degli standard per evitare i problemi di portabilità, poichè nelle varie implementazioni di Linux i data types usati per rappresentare le informazioni potrebbero essere diversi.

# LEZIONE 1 - Filsystem

**System call della libreria: `fcntl.h`.**

- ➔ `int open(const char *pathname, int flags, ... /*mode_t mode */)`: ritorna il file descriptor o -1 in caso di errore

La system call apre un file esistente oppure ne crea uno nuovo e poi lo apre. Se la open ha successo, ritorna un file descriptor utilizzato per referenziare il file. L'argomento `flags` è una bit mask per specificare la modalità di accesso al file.

| Flag | Descrizione |
|---|---|
| *O_RDONLY*<br>*O_WRONLY*<br>*O_RDWR* | Open for reading only<br>Open for writing only<br>Open for reading and writing |
| *O_TRUNC*<br>*O_APPEND* | Truncate existing file to 0 length<br>Writes are always appended to end of file |
| *O_CREAT*<br>*O_EXCL* | Create file if it doesn't already exist<br>With O_CREAT, ensure that creates the file |

Quando viene creato un nuovo file, viene considerato anche il parametro `mode`, che è una bit mask che indica i permessi per il file.

| Flag | Descrizione |
|---|---|
| *S_IRWXU*<br>*S_IRUSR*<br>*S_IWUSR*<br>*S_IXUSR* | User has read, write and execute permission<br>User has read permission<br>User has write permission<br>User has execute permission |
| *S_IRWXG*<br>*S_IRGRG*<br>*S_IWGRG*<br>*S_IXGRG* | Group has read, write and execute permission<br>Group has read permission<br>Group has write permission<br>Group has execute permission |
| *S_IRWXO*<br>*S_IROTH*<br>*S_IWOTH*<br>*S_IXOTH* | Other has read, write and execute permission<br>Others has read permission<br>Others has write permission<br>Others has execute permission |

**NOTA:** la **umask** (user file-creation mask) è un attributo che specifica quali permessi dovrebbero venire sempre assegnati al momento della creazione di nuovi file e solitamente ha il valore **022** (**--- -w- -w-**).

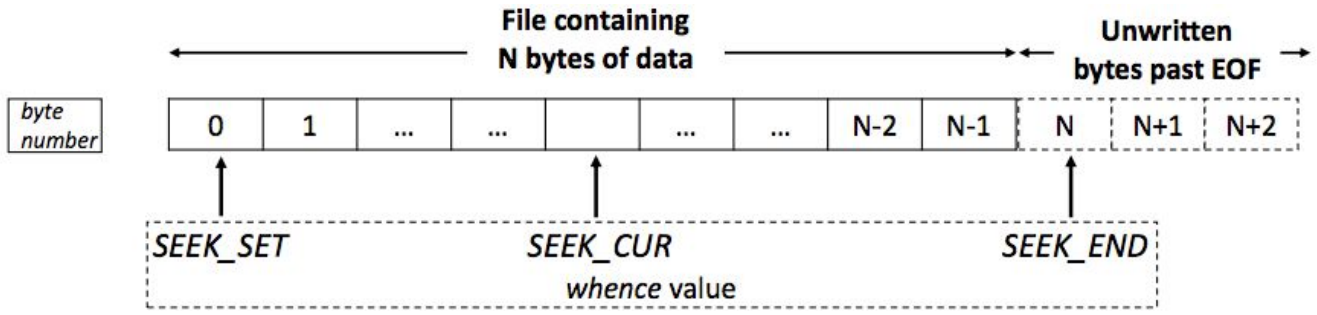
➔ `ssize_t read(int fd, void *buf, size_t count)`: ritorna il numero di bytes letti, o -1 in caso di errore. Se ritorna 0 = **EOF**

Il parametro `count` specifica il massimo numero di byte da leggere dal file descriptor fd. Il parametro `buf` contiene l'indirizzo di memoria in cui devono venire salvati i dati letti.
**NOTA**: se si fa la read da STDIN, in fondo c'è il carattere di new line '\n'.
**NOTA:** quando si esegue una **read**, il cursore si sposta di n posizioni (vedi esercizio 2 della seconda lezione)

**System call della libreria: `unistd.h`.**

➔ `ssize_t write(int fd, void *buf, size_t count)`: scrive sul file referenziato da fd il contenuto di buf, fermandosi dopo count bytes. Ritorna il numero di bytes scritti, o -1 in caso di errore

➔ `off_t lseek(int fd, off_t offset, int whence)`: consente di spostare il puntatore all'interno del file. Ritorna la posizione dell'offset risultante, o -1 in caso di errore




```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(fd4, -10, SEEK_CUR);
// 10th byte after the current offset location of the file.
off_t current = lseek(fd3, 10, SEEK_CUR);
```

- ➔ `int close(int fd)`: anche se i file descriptor vengono chiusi automaticamente quando il processo termina, è buona pratica chiuderli esplicitamente. Ritorna 0 in caso di successo, -1 in caso di errore.

- ➔ `int unlink(const char *pathname)`: rimuove un link, e se è l'ultimo link del file, rimuove anche il file stesso. Non funziona con le directory. Ritorna 0 in caso di successo, -1 in caso di errore.

## Attributi dei file

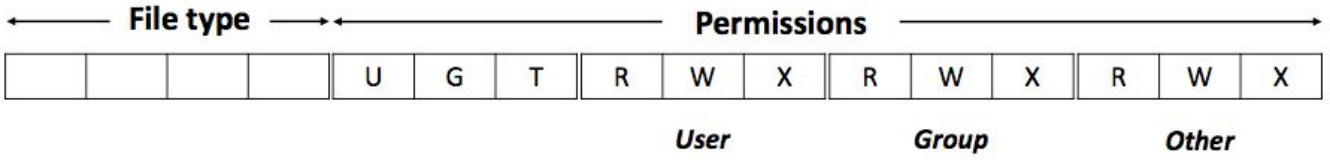
Un **inode** (*index node*) è una struttura data in un file system stile Unix che descrive un oggetto del file system come un file o una directory.
Ogni inode contiene gli attributi e la posizione del blocco nel disco dei dati dell'oggetto.
Gli oggetti del file system possono includere metadati (tempo di ultima modifica, ultimo accesso,..) oltre ai dati relativi al proprietario e alle autorizzazioni.

**Libreria:** `sys/stat.h`. Return 0 in caso di successo, -1 err.

- ➔ `int stat(const char *pathname, struct stat *statbuf)` → ritorna informazioni sul file
- ➔ `int lstat(const char *pathname, struct stat *statbuf)` → ritorna info su un link simb.
- ➔ `int fstat(int fd, struct stat *statbuf)` → è simile a *stat*, eccetto che il file è referenziato da un file descriptor invece del suo pathname

```c
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) links 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.
};
```

La combinazione di **st_dev** e **st_ino** identifica univocamente un file attraverso tutti i file system.
**st_mode** è una bit mask che ha il duplice scopo di identificare il tipo di file e specificare le autorizzazioni del file. I bit di questo campo sono disposti come segue:




Il **tipo di file** si può estrarre da `(st_mode & S_IFMT)` e comparando il risultato con un range di costanti.
Oppure si può usare una delle macro definite, passando come parametro st_mode.

| 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_ISFIFO()* | FIFO or pipe |
| *S_IFSOCK* | *S_ISSOCK()* | Socket |
| *S_IFLNK* | *S_ISLNL()* | Symbolic link |

I bit segnati con **U** e **G** sono applicati per gli eseguibili.
- ➢ ***set-user-ID***: se impostato, l'ID utente effettivo del processo viene reso uguale al proprietario dell'eseguibile;
- ➢ ***set-group-ID***: se è impostato, l'ID gruppo effettivo del processo viene reso uguale al proprietario dell'eseguibile.

Il bit etichettato **T**, chiamato **Sticky-bit**, funge da flag di eliminazione limitata per la directory.
L'impostazione di questo bit su una directory significa che un processo senza privilegi può scollegare (unlink (), rmdir ()) e rinominare (rinominare ()) i file nella directory solo se ha l'autorizzazione di scrittura sulla directory e possiede il file o la directory

*Directory e file hanno lo stesso schema dei permessi, ma sono interpretati diversamente.*

| FILE | DIRECTORY |
|---|---|
| ➢ ***read***: il contenuto del file può essere letto<br>➢ ***write***: il contenuto del file può essere cambiato<br>➢ ***execute***: il file può essere eseguito | ➢ ***read***: il contenuto della directory può essere elencato<br>➢ ***write***: si possono creare e rimuovere file dalla directory<br>➢ ***execute***: è possibile accedere ai file all'interno della directory |

**NOTA**: quando si vuole accedere a un file, è richiesto il permesso execute su tutte le directory elencate nel pathname.

Per vedere i permessi dei file si può unire con una AND la **st_mode** e una delle seguenti costanti definite nella libreria sys/stat.h

| 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 |

Ci sono due system call per **cambiare i permessi** di un file:

- ➔ `int chmod(const char *pathname, mode_t mode)`: cambia i permessi passando il pathname come parametro
- ➔ `int fchmod(int fd, int mode)`: come chmod, ma invece del pathname ha come parametro il file descriptor

Return 0 se ha successo, -1 con errore

**Libreria:** `unistd.h`

- ➔ `int access(const char *pathname, int mode)`: return 0 se vengono concesse tutte le autorizzazioni, altrimenti -1

Questa system call controlla l'accessibilità di un file basato su utente e id gruppo reali del processo. Se **pathname** è un link simbolico, lo dereferenzia.
**mode** è una bit mask che consiste in una o più delle seguenti costanti.

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

## Operazioni sulle cartelle

Entrambe le system call ritornano `0 in caso di successo, -1 errore`

- ➔ `int mkdir(const char *pathname, mode_t mode)`: il pathname può essere relativo o assoluto. Se un file con quel pathname esiste già, la system call fallisce con errore **EEXIST**.
- ➔ `int rmdir(const char *pathname)`: la directory deve essere vuota. Se l'ultimo componente del pathname è un link simbolico, non viene dereferenziato ma risulta l'errore **ENOTDIR.**

**Librerie:** `sys/types.h, dirent.h`

- ➔ `DIR *opendir(const char *dirpath)`: ritorna un **directory stream handle**, NULL in caso di errore.
- ➔ `int closedir(DIR *dirp)`: chiude lo stream aperto, ritorna 0 se ha successo, -1 errore.
- ➔ `struct dirent *readdir(DIR *dirp)`: ritorna un puntatore alla struttura allocata che descrive la prossima voce della directory, altrimenti NULL se è la fine della directory o in caso di errore
- ➔

Ogni chiamata a **readdir** legge il file/directory successivo nello stream aperto, e ogni entry ha la struttura descritta a destra (**struct dirent**)

```
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.
    //...
}
```

Sono definite delle macro costanti per il valore di **d_type** che identifica il tipo di file.

| 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 |

# LEZIONE 2 - Processi

## Attributi di un processo

### Identificatore

**NOTA**: le prossime system call hanno sempre successo.

**Process Identifier**

- ➔ `pid_t getpid(void)`: ritorna il process ID the processo chiamante

Il **PID = Process ID** è un identificativo univoco associato ad un processo. Solo un processo ha il PID fisso: **init**, che è il processo root dell'albero dei processi del sistema operativo.
Nella shell, per vedere l'albero dei processi uso il comando `ps auxf`

**User-ID reale ed effettivo**

- ➔ `uid_t getuid(void)`
- ➔ `uid_t geteuid(void)`
- ➔ `gid_t getgid(void)`
- ➔ `gid_t getegid(void)`

Lo User ID (e group ID) **reale** è l'utente a cui appartiene il processo.
Lo User ID (e group ID) **effettivo** è utilizzato per determinare i permessi garantiti al processo quando prova a eseguire delle operazioni.
**NOTA**: se lo **sticky bit** non è settato, vengono assegnati i permessi dell'utente per fare le operazioni. Altrimenti, sono garantiti i permessi di owner.

**Ambiente (Process environment)**

Ogni processo ha associato un array di stringhe chiamato **environment list**, o semplicemente **environment**, che contiene le variabili d'ambiente. Ognuna di queste stringhe è una definizione nella forma `name = value`.
Quando un processo viene creato, eredita una copia delle variabili d'ambiente del padre.
Tale array è accessibile in due modi:

- ➔ `extern char **environ`
- ➔ `int main(int argc, char *argv[], char* env[])`

**NOTA**: **environ non è definito in nessun header, è definita nei codici sorgente del kernel. Quindi l'unico modo per accedervi è usare la direttiva ***extern***

Per accedere e gestire le variabili d'ambiente ci sono le seguenti system call:

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

- ➔ `char *getenv(const char *name)`: ritorna un puntatore al valore, NULL se non esiste
- ➔ `int setenv(const char *name, const char *value, int overwrite)`: 0 successo, -1 errore. Se overwrite = 0 e la variabile d'ambiente esiste già, non viene sovrascritta. Se overwrite !=, environment viene sempre cambiato.
- ➔ `int unsetenv(const char *name)`: 0 successo, -1 errore

## Working directory

- ➔ `char *getcwd(char *cwdbuf, size_t size)`: consente al processo di avere la current working directory. Il primo parametro indica dove salvare il risultato, il secondo indica la dimensione massima (solitamente si usa una costante PATH_MAX). Se il risultato supera size, ritorna NULL.
- ➔ `int chdir(const char *pathname)`: cambia la cwd del processo chiamante (pathname può essere relativo o assoluto)
- ➔ `int fchdir(int fd)`: come chdir, ma invece del pathname ha come parametro il file descriptor.

## File descriptor table

Ogni processo ha associata una tabella dei file descriptor. Ogni entry rappresenta una risorsa di input/output usata dal processo. La cartella `/proc/<PID>/fd` contiene un link simbolico per ogni entry della tabella dei file descriptor del processo.

**NOTA: ad ogni nuova entry della tabella dei file descriptor viene assegnato il più basso indice disponibile**

**Esempio**: se chiudiamo stdout, poi apriamo un file e facciamo una printf, il risultato non verrà stampato a video ma nel file che abbiamo aperto. Infatti, chiudendo stdout si è liberato l'indice 1, che viene poi assegnato al nostro file e printf lavora di default sul fd 1 perchè si aspetta che sia lo standard output.

- ➔ `int dup(int oldfold)`: prende come parametro un file descriptor aperto, e ritorna un nuovo descriptor che fa riferimento allo stesso file descriptor aperto. Errore = -1.

# Operazioni con i processi

## Terminazione

Un processo può essere terminato richiamando una delle seguenti system call, oppure tramite una **return** (esplicita o implicita)

- ➔ `void _exit(int status)`: il processo viene sempre terminato con successo. Per convenzione, lo stato 0 indica che il processo è terminato con successo.
- ➔ `void exit(int status)`:
  - ➢ provvede a chiamare gli exit handlers
  - ➢ esegue il flush degli stdio stream buffers
  - ➢ chiama _exit(), usando il valore di status passato come parametro

Un **exit handler** è una funzione che viene registrata durante la vita di un processo. Viene automaticamente chiamata durante il processo di terminazione tramite **exit()**.
Se vengono registrati più exit handler, vengono chiamati nell'**ordine inverso di registrazione**.

- ➔ `int atexit(void (*func)(void))`: aggiunge il puntatore della funzione passata come parametro alla lista delle funzioni che vengono chiamate alla terminazione.

## Creazione

- ➔ `pid_t fork(void)`: nel **padre** ritorna il process ID del figlio, errore = -1. Nel **figlio** ritorna sempre 0.

Questa system call **crea un nuovo processo**, il *figlio*, che è quasi un duplicato esatto del processo chiamante, il *padre*.
Dopo l'esecuzione della **fork()**, esistono due processi, ed in ogni processo l'esecuzione continua dal punto in cui la fork è ritornata.
Non è determinato quale dei due processi sia il prossimo ad essere schedulato.
Il figlio i duplicati dei file descriptor del padre e la memoria condivisa allegata.
**NOTA**: il figlio ha pid = 0, ma **è solo per poter distinguere il figlio dal padre**. Padre e figlio hanno due pid che sono diversi dal valore ritornato dalla fork().

Ogni processo ha un padre e per sapere il suo PID possiamo utilizzare la system call:

- ➔ `pid_t getppid(void)`: ha sempre successo, poichè tutti i processi hanno un padre. **init** è il padre di tutti. Se un processo figlio diventa **orfano** perchè il padre ha terminato, allora viene "adottato" dal processo init.

## Monitoring

- ➔ `pid_t wait(int *status)`: ritorna il pid del figlio che ha terminato, errore = -1.
  - ➢ se il processo chiamante non ha figli da attendere, ritorna -1 **errno = ECHILD**.
  - ➢ se nessun figlio ha ancora terminato, la wait **blocca** il processo chiamante finchè un figlio non termina. Se un figlio ha già terminato, allora la wait ritorna subito.
  - ➢ Se **status != NULL**, le informazioni sull'elemento figlio terminato vengono archiviate nella variabile intera a cui indica lo stato

**NOTA**: cosa succede ad un figlio che termina prima che il padre abbia la possibilità di eseguire una *wait* ?
Il **kernel** si occupa di questa situazione trasformando il processo terminato in un processo zombie. Questo significa che molte delle risorse del figlio vengono rilasciate al sistema, ma non tutte. Le uniche parti mantenute sono: PID, stato di terminazione, le statistiche di uso delle risorse.
Se il processo padre termina senza chiamare la wait, allora il processo zombie viene "adottato" da init, che eseguirà la call system wait un po' di tempo dopo.

- ➔ `pid_t waitpid(pid_t pid, int *status, int options)`: sospende l'esecuzione del processo chiamante finchè un figlio specificato dal parametro pid non cambia stato. Il valore di **pid** determina quale processo vogliamo aspettare.

Valori di `pid`:

- ➔ **pid ≥ 0**: aspetta il figlio che ha il PID uguale al parametro
- ➔ **pid = 0**: aspetta qualsiasi figlio nello stesso gruppo del processo chiamante
- ➔ **pid < -1**: aspetta qualsiasi figlio nel gruppo del processo |pid|
- ➔ **pid = -1**: aspetta qualsiasi figlio

Valori di `options` (OR pari a zero o più delle seguenti costanti):

- ➔ **WUNTRACED**: return quando un figlio viene stoppato da un **segnale** o termina
- ➔ **WCONTINUED**: return quando un figlio è stato ripreso (resumed) da un segnale di **SIGCONT**
- ➔ **WNOHANG**: se nessun figlio specificato da pid ha ancora cambiato stato, allora ritorna immediatamente, invece di bloccare (esegue una "**poll**"). In questo caso, il valore di ritorno della system call è 0.

Se viene applicata solo la costante 0, allora *waitpid* attende solo per i figli che terminano.

Parametro **pid**: ci permette di distinguere i seguenti eventi di un processo figlio:

1. Il processo è terminato con una _exit ( o exit ) ⇒ **terminazione normale**
   - ➢ La macro **WIFEXITED** ritorna true se il figlio ha terminato normalmente
   - ➢ La macro **WEXITSTATUS** ritorna lo stato di uscita del processo figlio
2. Il processo figlio è **terminato** da un **segnale**
   - ➢ La macro **WIFSIGNALED** ritorna true se il figlio è stato terminato da un segnale
   - ➢ La macro **WTERMSIG** ritorna il numero del segnale che ha causato la terminazione
   - ➢ `strsignal(int sig)`: è un metodo di **string.h** che ritorna una stringa che descrive il segnale
3. Il processo figlio è **stoppato** da un **segnale**
   - ➢ La macro **WIFSTOPPED** ritorna true se il processo figlio è stato stoppato da un sign.
   - ➢ La macro **WSTOPSIG(status)** ritorna il numero del segnale che ha stoppato il processo
4. Il processo è stato **ripreso** dopo un segnale **SIGCONT**
   - ➢ La macro **WIFCONTINUED** ritorna true se il figlio è stato ripreso da un segnale SIGCONT

## Esecuzione di programmi

La famiglia delle funzioni **exec** sostituisce l'immagine del processo corrente con l'immagine di un nuovo processo

- ➔ `int execl (const char *path, const char *arg, ...)`
- ➔ `int execlp(const char *path, const char *arg, ...)`
- ➔ `int execle(const char *path, const char *arg, ..., char *const envp[])`
- ➔ `int execv (const char *path, const char *argv[])`
- ➔ `int execvp(const char *path, const char *argv[])`
- ➔ `int execve(const char *path, const char *argv[], char *const envp[])`

**NOTA**: la lista degli argomenti deve terminare con un puntatore a **NULL**, e siccome queste sono **funzioni variadiche** (cioè può accettare un numero variabile di argomenti), deve essere fatto il cast sul puntatore ⇒ **(char *)NULL**.

- ✓ Il parametro di input del programma punta a un **eseguibile**
- ✓ Liste e array terminano sempre con un **NULL pointer**
- ✓ Per convenzione, il primo elemento di **argv** è il nome del programma
- ✓ Tutte le funzioni exec **non hanno risultato in caso di successo**

| funzione | path | argomenti (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 |

**NOTA**: se c'è la **l** ⇒ lista argomenti, **v** ⇒ array argomenti. **p** ⇒ filename, altrimenti pathname. **e** ⇒ array environ, altrimenti caller's environ.

- ✓ **path**: *pathname* indica il path assoluto dell'eseguibile, mentre *filename* il nome dell'eseguibile, che deve trovare nella lista delle cartelle specificate nella variabile d'ambiente PATH
- ✓ **argv**: definisce gli argomenti da linea di comando del programma
- ✓ **envp**: array di puntatori a string (name = value) che definisce l'ambiente del programma

**Esempio execve**

```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");
}
```

# LEZIONE 3 - Interprocess communication (IPC): Signal, PIPE e FIFO

**IPC**: queste strutture/funzioni servono a far sì che processi appartenenti allo stesso gruppo o processi che sono slegati fra loro possano comunicare e scambiarsi informazioni

## Segnali

### Concetti fondamentali

Un **segnale** è una notifica ad un processo che si è verificato un certo evento.

I segnali interrompono il normale flusso di esecuzione di un programma e, nella maggior parte dei casi, non è possibile predire esattamente quando avverrà un certo segnale, **non è deterministico.**

Un segnale viene generato da un evento e viene poi consegnato ad un processo. Nell'intervallo di tempo da quando un segnale viene generato a quando viene consegnato, il segnale viene definito ***pending***.

Normalmente, un segnale nello stato *pending* viene consegnato al processo il prima possibile (a seconda dello scheduler) o immediatamente se il processo è già in esecuzione.

Quando un segnale viene consegnato, possono essere eseguite varie azioni:

- ➔ **killed**: il processo viene terminato
- ➔ **stopped**: il processo viene sospeso
- ➔ **resumed**: il processo viene ripreso dopo essere stato sospeso in precedenza
- ➔ **ignored**: il segnale viene ignorato, viene scartato dal kernel e il processo non sa nemmeno l'esistenza di questo segnale.
- ➔ Il processo esegue un ***signal handler***, una funzione scritta dal programmatore che deve eseguire determinati task in risposta al segnale.

### Tipologia di segnali

La lista completa di segnali disponibili in Linux può essere recuperata con il comando `man 7 signal`

- ➔ **SIGTERM**: viene consegnato al processo per farlo terminare il modo **sicuro**.
- ➔ **SIGINT** : termina un processo ("interrupt process"). Ctrl+C
- ➔ **SIGQUIT**: termina un processo e viene generato un core dump, che può essere utilizzato in seguito ad esempio per operazioni di debugging
- ➔ **SIGKILL**: **termina sempre** il processo. Non può essere bloccata, ignorata o gestita con un handler (spietato cit. Chicco)
- ➔ **SIGSTOP**: **ferma sempre** il processo. Non può essere bloccata, ignorata o gestita con un handler
- ➔ **SIGCONT**: fa ripartire un processo che era stato precedentemente bloccato
- ➔ **SIGPIPE**: viene generato quando un processo prova a scrivere su una pipe, però non c'è nessun file descriptor aperto in lettura su questa pipe, non c'è nessun processo che aspetta di leggere da questa pipe.
- ➔ **SIGALRM**: viene inviato ad un processo quando un real-time timer conclude il countdown
- ➔ **SIGURS1, SIGUSR2**: sono per gli sviluppatori, il kernel non genera mai questi segnali per un processo.

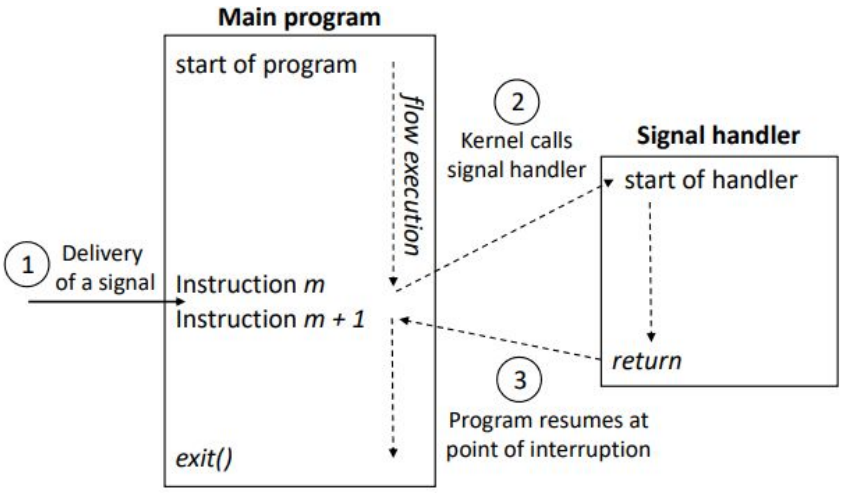
| name | number | can be caught? | default action |
|---|---|---|---|
| SIGTERM | 15 | yes | terminates a process |
| SIGINT | 2 | yes | terminates a process |
| SIGQUIT | 3 | yes | dumps + terms a process |
| SIGKILL | 9 | no | terminates a process |
| SIGSTOP | 17 | no | stops a process |
| SIGCONT | 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 |

## Signal handler (o signal catcher)

Un signal handler è una funzione che viene chiamata appena lo specifico segnale a cui è associata viene consegnato ad un processo.

➔ `void sigHandler(int sig)`

Questa funzione non ritorna niente e ha come parametro un intero (il segnale). Quando il signal handler viene chiamato dal kernel, il parametro sig viene settato con il numero del segnale consegnato al processo.
Un handler può gestire anche più segnali.




Mediante la system call **signal** si può cambiare il signal-handler di default per uno specifico segnale in un processo.

```
#include <signal.h>
typedef void (*sighandler_t)(int): interfaccia di un signal handler
sighandler_t signal(int signum, sighandler_t handler)
```

La funzione *signal* ritorna il segnale in caso di successo, oppure SIG_ERR in caso di errore.

**signum**: identifica il segnale per il quale si vuole specificare un gestore, l'handler.
L'**handler** può essere di diversi tipi:

- ➢ una funzione definita dall'utente
- ➢ la costante **SIG_DFL**, usa l'handler di default, resetta qualsiasi precedente cambio
- ➢ la costante **SIG_IGN**, ignora la consegna di quel segnale

**NOTE**:

- ➢ SIGKILL E SIGSTOP non si possono catturare
- ➢ I segnali sono eventi asincroni, non si può predire quando arrivano
- ➢ Quando un signal handler viene invocato, il segnale in causa viene bloccato automaticamente (vuol dire che non può essere generato un altro segnale dello stesso tipo finchè non viene sbloccato). Viene sbloccato quando il signal handler ritorna alla normale esecuzione del programma
- ➢ Se un segnale bloccato viene generato più volte, quando viene sbloccato viene consegnato una sola volta al segnale (della coda, ne viene considerato solo uno)
- ➢ L'esecuzione di un signal handler può essere interrotta da un altro segnale non bloccato
- ➢ Le disposizioni dei segnali sono **ereditati** da padre a figlio

## Waiting for a signal

Libreria: `<unistd.h>`

➔ `int pause()`

Sospende l'esecuzione di un processo finchè non viene interrotto dall'esecuzione di un signal handler. Ritorna sempre -1 con errno settato a EINTR.

➔ `unsigned int sleep(unsigned int seconds)`

Sospende l'esecuzione di un processo per un numero di secondi specificato nel parametro o finchè un segnale non viene catturato (e interrompe la call alla sleep).
Ritorna 0 se completa normalmente, o il numero dei secondi non attesi se termina prima.

## Inviare un segnale

Libreria: `<signal.h>`

- ➔ `int kill(pid_t pid, int sig)`, ritorna 0 se ha successo, -1 con errore

La system call kill consente di inviare un segnale ad un altro processo
L'argomento **pid** identifica uno o più processi a cui inviare il segnale **sig**.

- ➢ **pid > 0**: si vuole inviare un segnale ad un processo con PID = pid
- ➢ **pid = 0**: si vuole inviare il segnale a tutti i processi appartenenti al gruppo del processo chiamante, compreso il chiamante stesso
- ➢ **pid < 0**: il segnale viene inviato a tutti i processi appartenenti ad un process group identificato dal valore assoluto di pid
- ➢ **pid = -1**: il segnale viene inviato a tutti i processi a cui il processo ha il permesso di inviare un segnale, escluso init e il processo stesso

**Processo group:** è un gruppo di processi che, generalmente, sono stati generati dallo stesso padre. Ci sono delle casistiche in cui il processo group id è diverso.

- ➔ `unsigned int alarm(unsigned int seconds)`, ha sempre successo, ritorna il numero di secondi rimanenti che sarebbero serviti per far completare il timer precedente (o 0 se non c'erano allarmi impostati in precedenza)

Gli alarm sono dei segnali che possono essere installati tramite la system call alarm che, dopo un certo tempo, viene inviato un segnale SIGALRM al processo.
Impostando un timer con alarm, sovrascrive il timer precedente
Possiamo disabilitare un timer esistente usando la call **alarm(0)**.

## Settare e bloccare un segnale

Il tipo di dato **sigset_t** rappresenta un insieme di segnali.
Le seguenti funzioni ritornano entrambe 0 con successo e -1 con errore

- ➔ `int sigemptyset(sigset_t *set)` ⇒ inizializza un insieme di segnali vuoto
- ➔ `int sigfillset(sigset_t *set)` ⇒ inizializza un insieme che contiene tutti i segnali

Dopo l'**inizializzazione**, i segnali individuali possono essere aggiunti o rimossi da un set.
0 successo, -1 errore.

- ➔ `int sigaddset(sigset_t *set, int sig)` ⇒ aggiunge un segnale ad un set
- ➔ `int sigdelset(sigset_t *set, int sig)` ⇒ rimuove un segnale da un set

Con la seguente system call è possibile verificare l'appartenenza di un segnale ad un set
Ritorna 1 se il segnale appartiene al set, altrimenti 0

- ➔ ~~`int sigdelset(sigset_t *set, int sig)`~~

Per ogni processo, il kernel mantiene una **signal mask**, vale a dire un insieme di segnali per i quali la consegna al processo è attualmente bloccata (tutti i segnali per un determinato processo). Se un segnale bloccato viene inviato (non ancora consegnato) ad un processo, la consegna del segnale viene ritardata fino a quando tale segnale non viene sbloccato, cioè fino a quando quel segnale non viene rimosso dalla signal mask del processo.

La seguente system call consente di aggiungere o rimuovere esplicitamente un segnale dalla signal mask. Return 0 in caso di successo, -1 errore

- ➔ `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)`

L'argomento **how** determina le modifiche che la funzione applica alla signal mask:

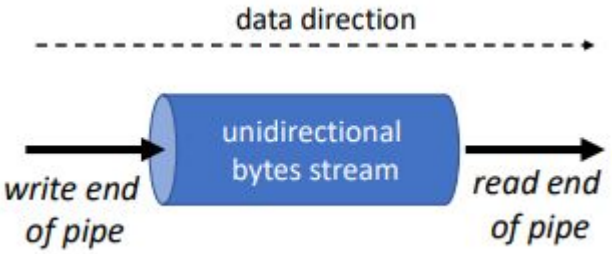
- ➢ **SIG_BLOCK**: i segnali specificati nel signal set (*set) sono aggiunti alla signal mask
- ➢ **SIG_UNBLOCK**: i segnali specificati nel *set sono rimossi dalla signal mask
- ➢ **SIG_SETMASK**: *set viene assegnato alla signal mask

In ogni caso, se *oldset* non è NULL, viene ritornata la signal mask precedente in *oldset
Se *set è NULL, viene ritornata la signal mask e *how* viene ignorato.

# Pipes

## Concetti fondamentali

Una **pipe** è un flusso di dati **sequenziale** e **unidirezionale** (byte stream) che consente ai processi di scambiarsi bytes.




All'interno delle pipe non c'è un concetto di messaggio o di limite di messaggio. Il processo legge dalla pipe blocchi di dati di qualsiasi grandezza, indipendentemente dalla grandezza dei blocchi scritti.
Sono pensate per la comunicazione tra **processi in relazione fra loro.**

La read si comporta in modo particolare.
Quando si legge da una pipe vuota, la read diventa bloccante (blocca l'esecuzione del processo) fino a quando almeno un byte non viene scritto nella pipe o viene inviato un segnale di non-terminazione (errno EINTR).
Se la parte di scrittura della pipe è chiuso, allora un processo che prova a leggere dalla pipe vedrà un carattere di EOF appena proverà a leggere, ma alla fine dei byte che sono contenuti nella pipe.

Se siamo i lettori e la pipe è vuota, ci blocchiamo fino a quando...
Se il lato di scrittura è chiuso, il processo che legge continua a leggere finche non finisce tutti i caratteri, e come ultimo carattere trova un EOF.

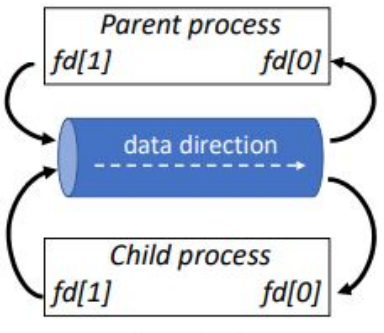
**Lato scrittore**
Una scrittura su una pipe può essere bloccante se non c'è spazio sufficiente nella pipe per completare l'operazione atomicamente.
Si blocca quando c'è spazio o arriva un segnale di non-terminazione (errno EINTR).
Scrivere blocchi di dati più grandi di PIPE_BUF potrebbero essere divisi in segmenti di grandezza arbitraria.

## Creazione e uso delle pipe

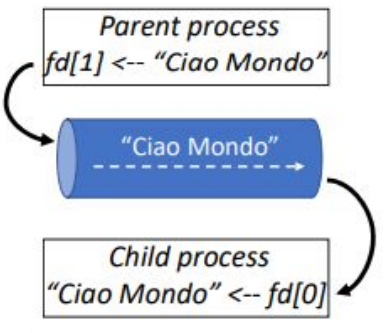
Libreria: `<unistd.h>`
➔ `int pipe(int fildes[2]);` return success = 0, error = -1
La system call **pipe** consente di creare una nuova pipe. Ha come parametro un array di due elementi interi. Se la chiamata alla pipe ha successo, ritorna i file descriptor del lato di scrittura e del lato di lettura della pipe:
**filedes[0]**: lato di lettura della pipe
**filedes[1]**: lato di scrittura della pipe

Normalmente, le pipe vengono usate per permettere la comunicazione tra processi correlati. Per connettere due processi usando una pipe, **seguiamo la pipe call con una call alla fork**

Nell'esempio a destra, sia padre che figlio hanno aperti entrambi i file descriptor, e attenzione, sono duplicati. Esempio: se chiudo fd[0] nel padre, nel figlio è ancora aperto. **NOTA:** Padre e figlio chiudono rispettivamente il file descriptor che non gli interessa.



After fork()



After closing unused descriptors

## FIFO (pipe nominali)

Una FIFO è uno stream di byte che consente ai processi di scambiare dati.
La differenza principale tra FIFO e PIPE è che una FIFO a un nome all'interno del file system (ha un file associato), ed è aperta e eliminata nello stesso modo di un file regolare.
Come nelle pipe, i dati vengono letti nello stesso ordine in cui vengono scritti.
Sono pensate per la comunicazione tra **processi non non in relazione fra loro.**

## Creazione, apertura e uso di FIFO

Libreria: `<unistd.h>`

➔ `int mkfifo(const char *pathname, mode_t mode)`

Il pathname specifica dove la FIFO è creata. Come in un file normale, il parametro mode specifica i permessi per la FIFO.
Dopo che una FIFO è stata creata, qualsiasi processo può aprirla.

➔ `int open(const char *pathname, int flags)`

Per aprire una FIFO, si usa una normale system call open.

L'uso ragionevole di una FIFO è avere un processo che legge e un processo che scrive.
Con le FIFO, il punto di sincronizzazione tra due processi è la open, la **open è bloccante.**
Le FIFO possono essere utilizzate per sincronizzare due processi.
Il processo che legge, se nessuno ha ancora scritto, si blocca.
Un processo che apre una FIFO in scrittura e prova a scrivere, rimane bloccato finchè un altro processo non apre la FIFO in lettura.

# LEZIONE 4 - IPC: System V e Message Queue

## System V

Le ICP **System V** si riferiscono a tre differenti meccanismi per la comunicazione interprocesso:

- ➢ ***Message queue***: le code di messaggi possono essere utilizzate per passare messaggi tra i processi. Sono simili alle pipe, ma ha due importanti differenze:
  - ✓ i messaggi hanno dei '**confini**', hanno una grandezza ben definita
  - ✓ ogni messaggio include un campo intero per il **tipo**
- ➢ ***Semaphores***: i semafori consentono ai processi di sincronizzare le azioni. Un semaforo è un valore mantenuto dal kernel, che viene opportunamente modificato dai processi del sistema prima di eseguire alcune azioni critiche
- ➢ ***Shared memory***: la memoria condivisa consente a più processi di condividere la loro regione di memoria

Le interfacce delle System V hanno delle caratteristiche comuni:

| 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 |

Ogni meccanismo di system v ha associata una call system ***get***, che è analoga alla system call **open**.
Data una **key** intera (analoga al filename), la get può o **creare** una nuova IPC e poi ritornare il suo identificatore univoco, oppure **ritornare** l'identificatore di una IPC esistente.

L'**identificatore** di una IPC è analogo al *file descriptor* e viene utilizzato in tutte le chiamate successive per fare riferimento alla IPC.

## Creazione e apertura di una System V

Esempio della **creazione** di una message queue.

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

Come in tutte le chiamate *get*, la **key** è il primo argomento, e **deve essere univoca**, serve a identificare l'oggetto IPC.
L'**identificatore** dell'IPC che viene ritornato è un codice univoco che identifica l'oggetto IPC nel sistema.
Facendo un confronto con la system call **open**: **key = filename, id = file descriptor**.

Le chiavi sono valori interi rappresentati mediante il tipo di dato **key_t**.
**NOTA**: La chiamata alla get traduce una chiave nel corrispondente identificatore intero.
Il **problema** è che potremmo usare una chiave già utilizzata per un altro oggetto IPC.

## Definizione della Key (in modo univoco)

### IPC_PRIVATE

Quando viene creato un nuovo oggetto IPC, la key può essere specificata come **IPC_PRIVATE**. In questo modo, il problema di trovare una chiave univoca viene delegato al kernel.
Questa tecnica è molto utile nelle *applicazioni multiprocesso*, dove il processo padre crea gli oggetti IPC prima di fare una fork(), con il risultato che il figlio eredita l'identificatore dall'oggetto IPC.
**NOTA:** IPC_PRIVATE **non genera** una key_t, è essa stessa una key_t. Quindi, l'IPC Object **non è privato**, ma il kernel genera una chiave univoca e garantisce che nessun processo avrà la stessa chiave. Per due processi che comunicano tramite un IPC Object, devono averlo ereditato dallo stesso processo chiave. Quindi, processi che non hanno relazioni tra loro (padre-figlio) non possono usare la IPC_PRIVATE key per accedere allo stesso oggetto.

### FTOK

La funzione **ftok** (file to key) converte un **pathname** e un **proj_id** (project identifier) in una chiave System V.
Libreria: `<sys/ipc.h>`

```
➔ key_t ftok(char *pathname, int proj_id)
```

Ritorna la **key** in caso di successo, altrimenti -1 (controllare errno)
Il **pathname** deve fare riferimento ad un file che deve essere **esistente** e **accessibile**.
**proj_id** deve essere un valore **diverso da 0**, è un int ma vengono utilizzati solo gli ultimi 8 bit.
Tipicamente il pathname fa riferimento a file, o directory, creati dall'applicazione.

## Struttura dati associata

Il kernel mantiene una struttura dati associata per ogni istanza di un oggetto IPC System V.
Oltre ai dati specifici per il tipo di oggetto IPC, ogni struttura dati associata include la sottostruttura **ipc_perm** che contiene le autorizzazioni concesse.

```
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 */
};
```

**NOTA**: cuid e cgid sono campi immutabili.
**NOTA:** normalmente, quello che interessa per gli oggetti IPC sono i permessi **read e write** (non serve il permesso di esecuzione, viene ignorato)

Esempio di utilizzo della ipc_perm con la **msgctl** (cambia il possessore della message queue, il codice sarà similare anche per semafori e memoria condivisa)

```
struct msqid_ds msgq;
// get the data structure of a message queue from the kernel
if (msgctl(msqid, IPC_STAT, &msgq) == -1)
    errExit("msgctl get failed");
// change the owner of the message queue
msgq.msg_perm.uid = newuid;
// update the kernel copy of the data structure
if (msgctl(msgid, IPC_SET, &msgq) == -1)
    errExit("msgctl set failed");
```

## Comandi IPC

`ipcs`: ci consente di ottenere le informazioni relative agli oggetti IPC nel sistema.
Di default, questo comando mostra 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
```

`ipcrm`: consente di rimuovere oggetti IPC dal sistema.
Per rimuovere una **message queue**: **-q** (per usare l'identificatore) o **-Q** (per usare la chiave)
Per rimuovere un segmento di **shared memory**: **-m** o **-M**
Per rimuovere un semaforo: -s o -S.

## Message Queue

La system call **msgget** crea una nuova message queue, oppure ritorna l'identificatore di una esistente.

➔ `int msgget(key_t key, int msgflg)`

Ritorna l'identificatore della message queue in caso di successo, altrimenti -1.
L'argomento key è una **chiave IPC**.
**msgflg** è una bit mask che specifica i permessi (come il parametro mode nella open) associati ad una nuova message queue, o per verificare se una certe queue esiste. I flag possono essere messi in ORed (|):

- **IPC_CREAT**: se non esiste nessuna message queue con la key specificata, ne crea una nuova
- **IPC_EXCL**: insieme a IPC_CREAT, la msgget fallisce se esiste una message queue con la key specificata

Esempio di **creazione di una message queue**:

```
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_CREATE | S_IRUSR | S_IWUSR);

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

Un messaggio in una message queue ha **sempre** la seguente struttura:

```
struct msgget{
long mtype;   /* > 0
char mtext[]; /* body: è il corpo del messaggio, si può mettere quello che si vuole
}
```

**NOTA**: la prima parte del messaggio contiene il tipo del messaggio, e deve essere **un intero long maggiore di 0**.
Il resto del messaggio (**body**) è una struttura definita dal programmatore di lunghezza e contenuto arbitrari (quindi non obbligatoriamente un array di caratteri) e può anche essere omesso se non necessario.

**INVIARE I MESSAGGI**

➔ `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`
Ritorna 0 successo, -1 errore.
**msqui**: identificatore della message queue
**msgp**: indirizzo che punta ad una message structure
**msgsz**: specifica il numero di byte contenuti nel campo **mtext** (il body) del messaggio (**NON** deve comprendere mtype).
**msgflg**: può essere **0** oppure il flag **IPC_NOWAIT**

- normalmente, se una message queue è full, la message send è **bloccante**, blocca finchè non c'è abbastanza spazio disponibile per poter mettere il messaggio nella queue
- se è specificato il flag IPC_NOWAIT, se la coda è piena, la message send ritorna immediatamente con l'errore **EAGAIN** (non ci sono dati disponibili adesso, riprova dopo)

Vedi **esempi** slide 23/24.
**NOTA: size_t mSize = sizeof(struct mymsg) - sizeof(long)**

## RICEVERE I MESSAGGI

➔ `size_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg)`

Ritorna il numero di byte copiati in msgp in caso di successo, o -1 con errore.

**NOTA:** quando un messaggio viene letto, viene rimosso dalla coda

**msqid**: identificatore della message queue.

**msgp** è il buffer dove salvare il messaggio, mentre **msgsz** il massimo spazio disponibile nel campo mtext del buffer di lettura. La struttura del messaggio che si utilizza nella lettura non deve essere per forza uguale a quella della scrittura.

**msgtype**: seleziona il messaggio dalla message queue nel seguente modo:

- ➢ **uguale a 0**: **il primo** messaggio della queue viene rimosso e ritornato al processo chiamante
- ➢ **maggiore di 0**: il primo messaggio della queue che ha **mtype = msgtype** viene rimosso e ritornato al processo chiamato
- ➢ **minore di 0**: il primo messaggio con **mtype <= |msgtype|** viene rimosso e ritornato al processo chiamante.

**msgflg**: bitmask messe in or fra di loro:

- ➢ **IPC_NOWAIT**: di default, se non ci sono messaggi che rispettano le richieste, la **msgrcv** è bloccante e si blocca finchè non ci sono messaggi disponibili. Con il flag **IPC_NOWAIT** ritorna subito con errore **ENOMSG**
- ➢ **MSG_NOERROR**: di default, se la dimensione di mtext è più grande dello spazio disponibile, la **msgrcv** fallisce. Con il flag **MSG_NOERROR**, invece di rimuovere il messaggio dalla coda, va a troncare il campo mtext a msgsz bytes e ritorna al chiamante. Consente quindi di leggere il messaggio pezzo per pezzo, se non c'è abbastanza spazio.

## MESSAGE QUEUE CONTROL OPERATIONS

➔ `int msgctl(int msqid, int cmd, struct msqid_ds *buf)`

Effettua operazioni di controllo sulla message queue, ritorna 0 in caso di successo, altrimenti -1.

**msqid**: identificatore della message queue.

**cmd**: specifica le operazioni da fare sulla coda:

- ➢ **IPC_RMID**: **rimuove** immediatamente la message queue. Tutti i messaggi non letti sono persi, e i lettori/scrittori bloccati vengono svegliai con **errno** impostato a **EIDRM**. Per questa operazione, l'argomento buf viene ignorato.
- ➢ **IPC_STAT**: mette una **copia** della struttura dati msqid_ds associata alla message queue nel buffer puntato dall'argomento buf. **nota** che è una copia, quindi le modifiche fatte sulla copia non vanno ad influire sull'originale.
- ➢ **IPC_SET**: **aggiorna** i campi selezionati della msqid_ds associata alla message queue usando i valori contenuti nel buffer puntato da buf

## 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() */
};
```

**Nota**: solo i campi **msg_perm** e **msg_qbytes** possono essere modificati.

# LEZIONE 5 - IPC: Semaphores and Shared Memory

## Semafori

### Creazione e apertura di un Semaphore Set

La system call **semget** crea un nuovo **semaphore set** o ritorna l'identificatore di uno esistente.

➔ `int semget(key_t key, int nsems, int semflg)`

Prende come parametri la **key**, **nsems** che indica il numero di semafori > 0 che contiene il set. **semflg** è una bit mask che specifica i permessi per un nuovo set di semafori oppure controlla i permessi di un set esistente.

Inoltre, in **smflg** possono essere messi in ORed (|) i flag:

- ➢ **IPC_CREAT**: se non è specificato nessun set di semafori con la key specificata, ne crea uno nuovo
- ➢ **IPC_EXCL**: insieme a IPC_CREAT, la semget fallisce se esiste un set di semafori con la chiave specificata

### SEMAPHORE CONTROL OPERATIONS

La system call **semctl** consente di eseguire una serie di operazioni di controllo su un set di semafori o su un semaforo singolo all'interno di un set.

➔ `int semctl(int semid, int semnum, int cmd, ... /* union semun arg */)`

Ritorna un numero intero non negativo in caso di successo, altrimenti -1.
**semid** è l'identificatore del set di semafori sul quale eseguire l'operazione.
**cmd** è il comando, alcune operazioni di controllo richiedono un ultimo argomento (**semun** = semaphore **union**).

### UNION SEMUN

Deve essere definita esplicitamente dal programmatore prima di chiamare la system call semctl.
Bisogna prendere il codice a destra e copiarlo nel proprio programma.

```
#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
```

## Semaphore Control Operations

**NOTA:** un set di semafori deve essere sempre inizializzato prima di essere usato
**NOTA**: una volta ritornato lo stato, il semaforo potrebbe averlo già cambiato
**Usage template**: `int semctl(semid, 0 /*ignored*/, cmd, arg)`

- ➢ **IPC_RMID**: consente di rimuovere immediatamente un semaphore set. Tutti i processi che erano bloccati vengono risvegliati (con errore **EIDRM**). L'argomento arg non è richiesto.
- ➢ **IPC_STAT**: mette una copia della struttura dati **semid_ds** associata al set di semafori nel buffer puntato da **arg.buf**
- ➢ **IPC_SET**: aggiorna i campi selezionati della struttura semid_ds usando i valori nel buffer puntato da arg.buf

```
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 */
};
```

**NOTA**: si possono modificare i campi **uid**, **gid**, **mode** della sottostruttura **sem_perm**

**Usage template**: `int semctl(semid, semnum, cmd, arg)`
- ➢ **SETVAL**: il valore del semaforo n-esimo (n = semnum) nel set viene inizializzato al valore specifico **arg.val**
- ➢ **GETVAL**: non è richiesto il parametro arg, e ritorna il valore del semaforo specificato nel set semid

**Usage template**: `int semctl(semid, 0 /*ignored*/, cmd, arg)`
- ➢ **SETALL**: inizializza tutti i semafori nel set usando i valori contenuti nell'array puntato da **arg.array**
- ➢ **GETALL**: recupera i valori di tutti i semafori e vengono salvati all'interno di **arg.array**

**Usage template**: `int semctl(semid, semnum, cmd, arg)`
- ➢ **GETPID**: ritorna il process ID dell'ultimo processo che ha eseguito una **semop** sul semaforo n-esimo
- ➢ **GETNCNT**: ritorna il numero di processi che in quel momento stanno aspettando che il semaforo n-esimo incrementi
- ➢ **GETZCNT**: ritorna il numero di processi che in quel momento stanno aspettando che il semaforo n-esimo arrivi a 0

## Semaphore Operations

La system call **semop** esegue una o più operazioni (wait(P) e signal(P)) sui semafori.
- ➔ `int semop(int semid, struct sembuf *sops, unsigned int nsops)`

Ritorna 0 in caso di successo.
L'argomento **sops** è un puntatore ad un array che contiene una sequenza ordinata di operazioni da eseguire **atomicamente**, e **nsops** (>0) è la dimensione dell'array.

Gli elementi di **sops** sono strutture con la seguente forma.

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

Il campo **sem_num** identifica il semaforo appartenente al set specificato sul quale si devono eseguire le operazioni.
**sem_op** specifica le operazioni da eseguire:
- ➢ **sem_op > 0**: il valore di sem_op viene **aggiunto** al semaforo n-esimo (n = sem_num)
- ➢ **sem_op = 0**: il valore del semaforo n-esimo **viene controllato** per verificare se è **uguale a 0**. Se non è zero, il processo chiamante è **bloccato** finchè il valore del semaforo non diventa 0.
- ➢ **sem_op < 0**: **decrementa** il valore del semaforo n-esimo con **sem_op**. Blocca il processo chiamante finchè il valore del semaforo non ha raggiunto un livello che permette di eseguire l'operazione senza avere risultato negativo

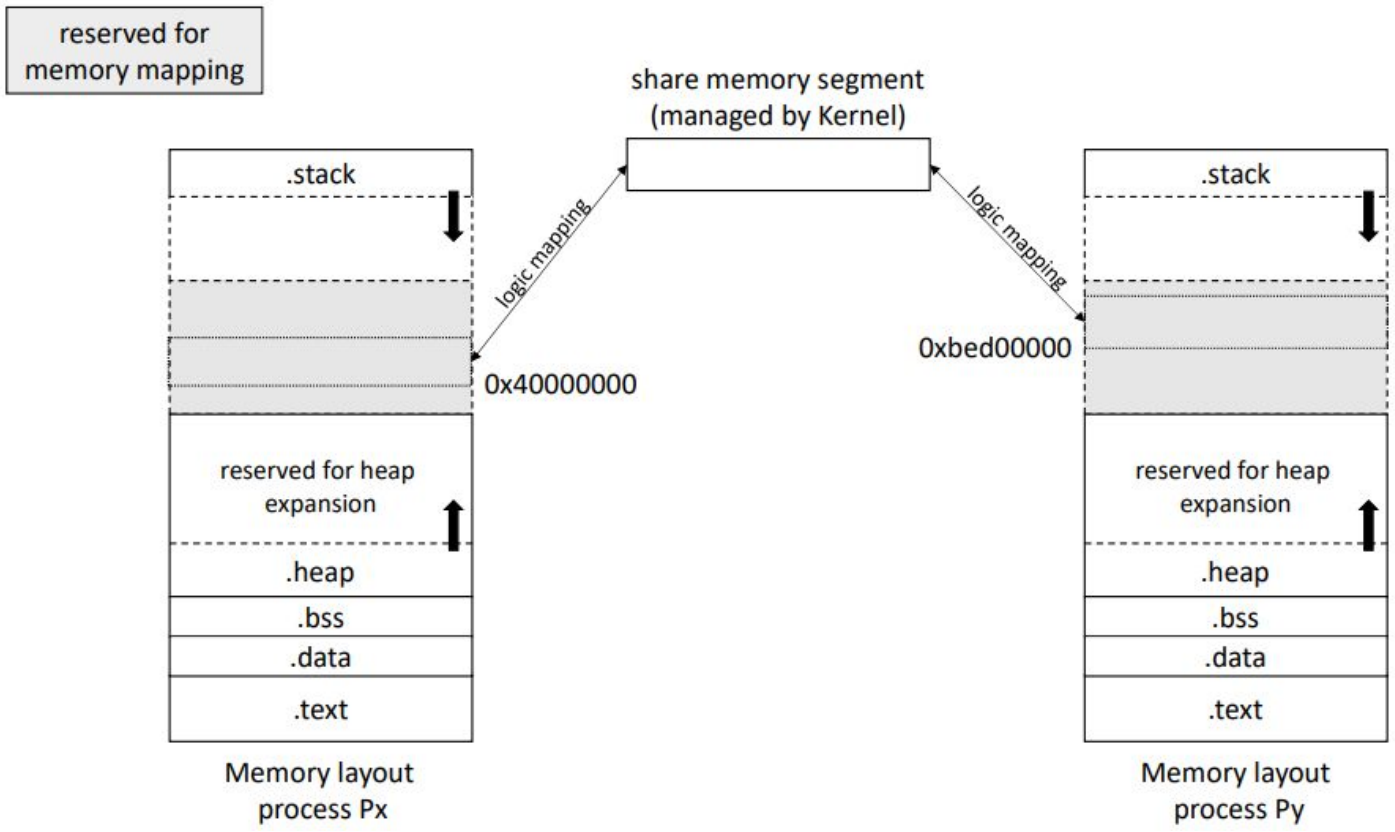
Quando la sem_op blocca, il processo rimane bloccato finchè:
- ➢ un altro processo modifica il valore del semaforo in modo tale che l'operazione richiesta può procedere
- ➢ un segnale interrompe la chiamata sem_op, viene ritornato l'errore **EINTR** (indica che la chiamata non si è sbloccata in modo naturale, ma è stata interrotta)
- ➢ il processo elimina il semaforo. In questo caso, sem_op fallisce con errore **EIDRM**

Possiamo prevenire il comportamento bloccante della semop specificando IPC_NOWAIT nel corrispondente campo sem_flg. In questo caso, se semop dovrebbe essere bloccata, fallisce con errore **EAGAIN**.

# Memoria condivisa

## CONCETTI FONDAMENTALI

La memoria condivisa è un segmento di memoria della memoria fisica gestita dal kernel, che permette a due o più processi di scambiare dati.
Una volta associata ad uno o più processi, la memoria condivisa fa parte effettivamente dello spazio virtuale di quel processo, senza nessun intervento del kernel richiesto.
Le scritture di dati nella memoria condivisa sono disponibili immediatamente per tutti i processi che condividono il segmento di memoria. Tipicamente, sono richiesti alcuni metodi di sincronizzazione per evitare l'accesso simultaneo di più processi alla memoria condivisa.




### Creazione e apertura della memoria condivisa

La system call **shmget** crea un nuovo segmento di memoria condivisa o ritorna l'identificatore di uno esistente. Il contenuto di un nuovo segmento creato è inizializzato a 0.
Libreria: `<sys/shm.h>`

➔ `int shmget(key_t key, size_t size, int shmflg)`

Ritorna l'identificatore del segmento di memoria, o -1 = errore.

Se stiamo usando la shmget per ottenere l'identificatore di un segmento esistente, l'argomento **size** non ha effetto sul segmento, ma deve essere <= della grandezza del segmento.
**shmflg** è una bit mask che specifica i permessi per un nuovo segmento di memoria o controlla i permessi di un segmento esistente. Inoltre, possono essere messi in ORed (|):

- ➢ **IPC_CREAT**: se non esiste nessun segmento con la chiave key, ne crea uno nuovo
- ➢ **IPC_EXCL**: insieme a IPC_CREAT, la shmget fallisce se esiste un segmento con la key specificata

**NOTA:** con la get chiediamo questo spazio/segmento di memoria al kernel, ma non **lo vediamo ancora**, non si può ancora utilizzare.
Questo perchè si deve "attaccare" → **attach** quel segmento di memoria all'area di memoria del processo.

## ATTACHING A SEGMENT

La system call **shmat** attacca il segmento di memoria condivisa allo spazio virtuale di indirizzamento del processo chiamante.

➔ `int *shmat(int shmid, const void *shmaddr, int shmflg)`

Ritorna l'indirizzo al quale la memoria condivisa è stata attaccata con successo, **altrimenti (void *)-1**

- **shmaddr NULL**: il kernel attacca il segmento in un indirizzo consono rispetto al segmento di memoria (shmaddr e shmflg sono ignorate)
- **shmaddr NOT NULL**: il segmento viene attaccato all'indirizzo specificato. Ma se:
  - **shmflg SHM_RND**: shmaddr viene arrotondato per difetto al più vicino multiplo della costante SHMLBA (shared memory low boundary address)

Normalmente, **shmaddr è NULL per diverse ragioni**:

- Se non usiamo NULL ma vogliamo scegliere noi l'indirizzo, questo riduce la portabilità di un applicazione. Un indirizzo valido in un implementazione UNIX potrebbe non è essere valido in un altra.
- Un tentativo di attaccare il segmento di memoria condivisa in un particolare indirizzo, fallirà se quell'indirizzo è già in uso

Inoltre, il flag **SHM_RDONLY** specifica di attaccare un segmento in sola lettura. Se shmflg non ha valore, la memoria condivisa viene collegata in modalità lettura e scrittura.
**NOTA:** conviene quindi usare shmaddr NULL per lasciare al kernel il compito di scegliere l'indirizzo di memoria

Un processo figlio eredita i segmenti di memoria condivisa del padre. La memoria condivisa fornisce un meccanismo di IPC facile e veloce per far comunicare fra loro padre e figlio.

## DEATTACHING A SEGMENT

Quando un processo non ha più bisogno dell'accesso ad un segmento di memoria, può essere **detached** con la system call **shmdt**.

➔ `int shmdt(const void *shmaddr)`

Durante una exec, tutti i segmenti di memoria condivisa sono detached.
Inoltre, i segmenti di memoria condivisa sono scollegati automaticamente quando termina un processo.

## SHARED MEMORY CONTROL OPERATIONS

La system call **shmctl** esegue un'operazione di controllo su un segmento di memoria condivisa.

➔ `int *shmctl(int shmid, int cmd, struct shmid_ds *buf)`

L'argomento **cmd** specifica l'operazione da eseguire:

- **IPC_RMID**: **marca** la memoria condivisa per l'eliminazione. Il segmento è rimosso appena tutti i processi eseguito il detach
- **IPC_STAT**: mette una **copia** della struttura dati **shmid_ds** nel buffer puntato da **buf**
- **IPC_SET**: **aggiorna** i campi selezionati della struttura dati **shmid_ds** usando i valori contenuti in **buf**

```
struct msqid_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
```

**NOTA**: sono importanti **shm_segsz** e **shm_nattach**.

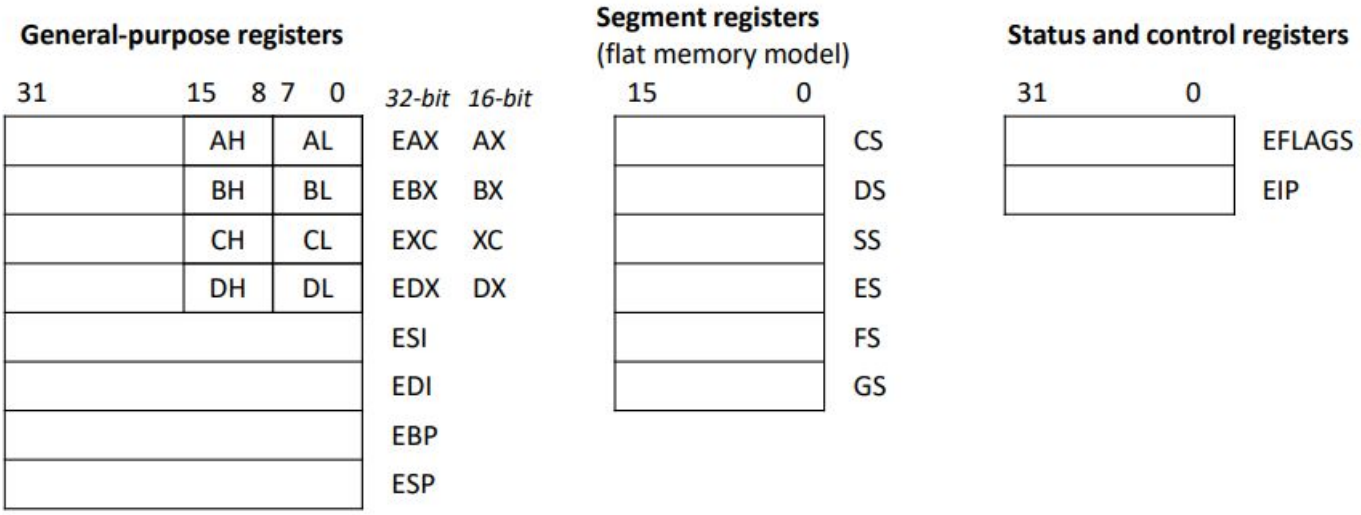
# LEZIONE 6 - Fundamental Concepts e Process Management
# Mentoring Operating System (MentOS)

## Concetti fondamentali

### Central Processing Unit (CPU)

Ci sono **TRE** tipi di registri:

- **general-purpose** data registers
- segment registers
- **status control** registers




#### General-purpose registers

Sono **otto** registri a **32 bit**. Sono usati per contenere gli operandi per operazioni logiche e aritmetiche, operandi per calcoli di indirizzi e puntatori di memoria → contengono tutti i dati che servono ai programmi per funzionare (operazioni matematiche, accessi a memoria,..).

- **EAX: accumulator**, contiene operandi, risultati di operazioni o dati
- **EBX: puntatore** ai dati nel segmento **DS**
- **ECX: contatore** per operazioni di loop
- **EDX: I/O pointer**
- **ESI: puntatore** a un dato nel segmento puntato dal registro **DS**
- **EDI: puntatore** a un dato nel segmento puntato dal registro **ES**
- **EBP (Base Pointer): puntatore** ai dati all'interno dello stack (nel segmento **SS**)
- **ESP (Stack pointer):** nel segmento **SS**

#### Status and control registers

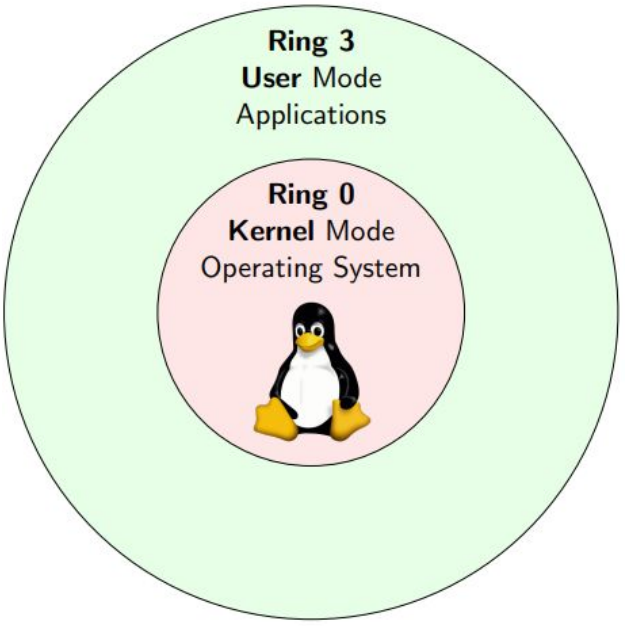
Sono **due** registri a **32 bit** usati per:

- **EIP**: **Instruction Pointer**
- **EFLAGS:** contiene tutte le flag di stato, controllo e di sistema che gestiscono l'esecuzione delle operazioni della cpu

| 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 |

## Livelli di privilegi

Ci sono **quattro** livelli di privilegio, **0 -> 3.**
In ogni momento, una CPU x86 esegue in uno specifico livello di privilegio, che determina **quali code possono essere eseguite, e quali no.**
I sistemi più moderni usano solo i livelli 0 e 3, come vediamo nell'immagine a destra.



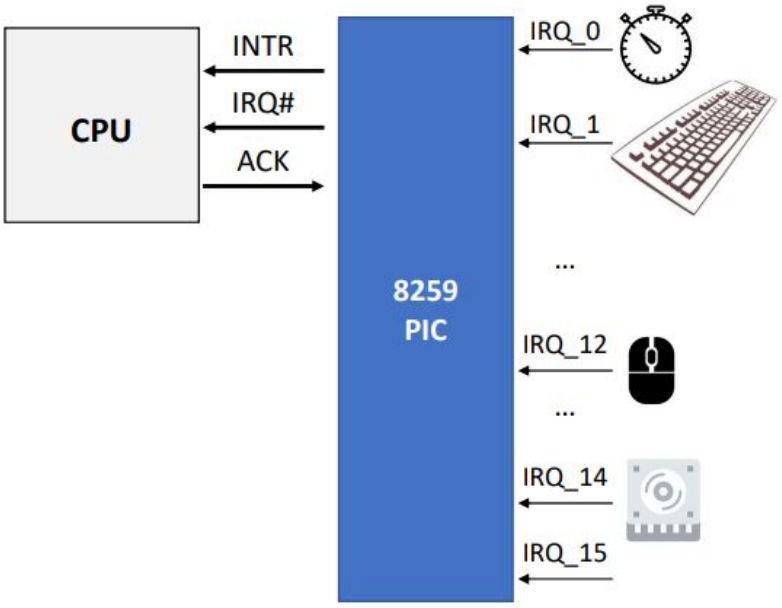

Le operazioni che possono essere eseguite quando la CPU è in **user mode**:

- aprire un file
- stampare sullo schermo
- allocare memoria **(NO, in kernel mode)**: si può chiedere al kernel di allocare la memoria

**NOTA**: ogni volta che la CPU cambia livello di privilegio, occorre un **CONTEXT SWITCH.**
Esempio: quando un utente preme un tasto sulla tastiera, ci sono 2 context switch.

## Programmable Interrupt Controller (PIC)

Un **programmable interrupt controller** è un componente che combina una serie di richieste interrupt in una o più linee di CPU.
Ci sono **16 linee di IRQ (Interrupt Request)**
**0** -> maggiore priorità
Nella linea 0 c'è il **timer**, che viene utilizzato moltissimo per gestire le tempistiche dell'intero kernel.

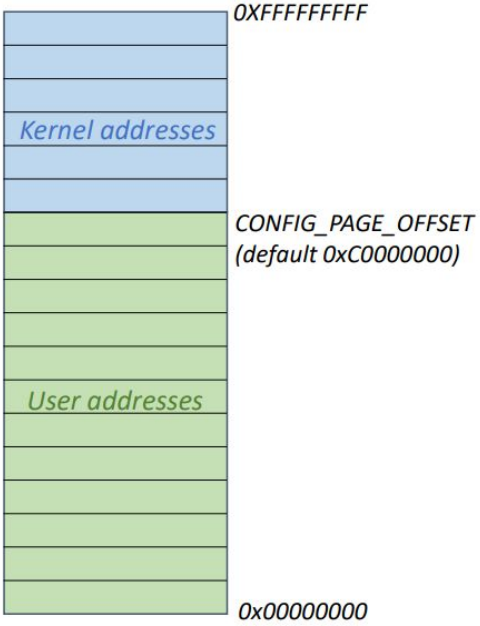



Il **TIMER** è un componente hw interno alla CPU, ha frequenza fissa e ad ogni fronte del segnale c'è un passaggio da user mode a kernel mode (fronte di salita) o viceversa (fronte di discesa)

In Linux, la frequenza del timer è di 100Hz, quindi la CPU esegue i processi in modalità utente per massimo 10 millisecondi, e in seguito il kernel riceve di nuovo il controllo della CPU.

## Organizzazione della memoria (32-bit system)

Il kernel usa la **memoria virtuale** per mappare gli indirizzi virtuali agli indirizzi fisici.
In particolare, la RAM è virtualmente divisa in **kernel space e user space**.
La CPU con privilegi **0** ha **visibilità dell'intera RAM.**
La CPU con privilegi **3** ha **visibilità solo dello user space.**



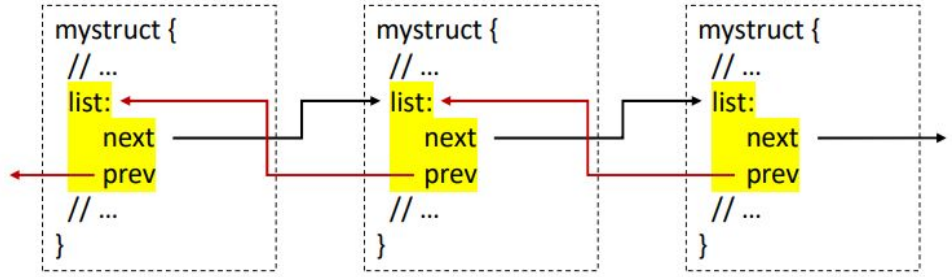
## Kernel doubly-linked list (CIRCOLARE)

Il kernel del sistema operativo, come molti programmi, ha spesso bisogno di mantenere una **lista di strutture dati**. Gli sviluppatori del kernel hanno creato un'**implementazione standard di una CIRCULAR, DOUBLY-LINKED LIST.**

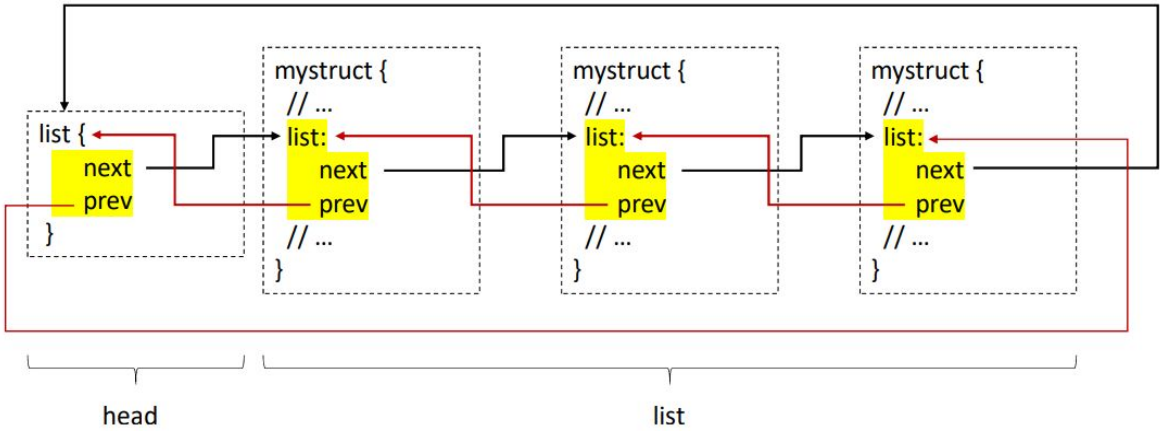
Un elemento di tipo **list_head_t** rappresenta un nodo di una lista. (nell'immagine è scritto sbagliato)

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

**NOTA:** Per usare questa struttura dati, è sufficiente dichiarare un campo di tipo **list_head** all'interno della nostra struttura per creare la lista (il campo di tipo list_head NON va dichiarato come puntatore)




**NOTA**: la **TESTA** della lista deve essere una struttura **list_head_t** autonoma. In una circular, doubly-linked list è sempre presente la testa.
Se la lista è **vuota**, allora esiste solo la testa.

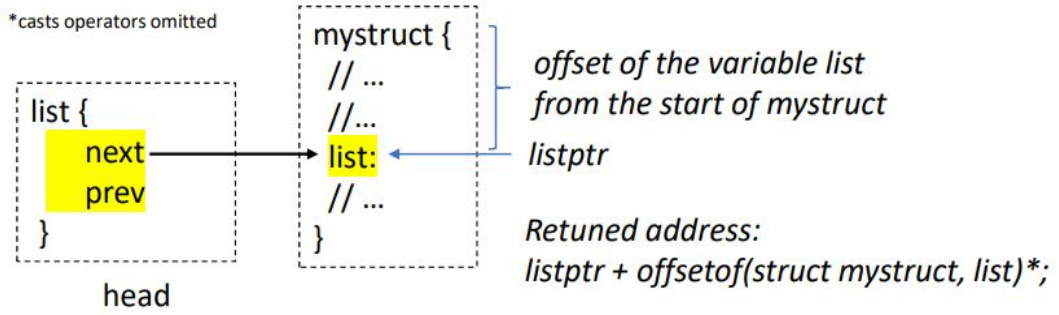



## Funzioni di supporto

https://repl.it/@galfurian/listhead

- ➔ `list_head_empty(list_head_t *head)`: ritorna un valore ≠ 0 se la lista data è vuota.
- ➔ `list_head_add(list_head_t *new, list_head_t *listnode)`: aggiunge la new entry subito dopo listnode
- ➔ `list_head_add_tail(list_head_t *new, list_head_t *listnode)`: aggiunge la new entry subito prima di listnode
- ➔ `list_head_del(list_head_t *entry)`: rimuove l'entry data
- ➔ `list_entry(list_head_t *ptr, type_of_struct, fieldname)`: ritorna la struttura con dentro **list_head**.
  - ➢ **ptr**: puntatore a un **list_head_t**
  - ➢ **type_of_struct**: è il nome del tipo della struttura che deve contenere un elemento della lista list_head_t
  - ➢ **field_name**: nome di list_head_t nella struttura

```
// 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);
```




➔ `list_for_each(list_head_t *ptr, list_head_t *head)`: itera ogni elemento della lista.
- ➢ **ptr**: free variable, puntatore di tipo list_head_t
- ➢ **head:** puntatore ad un head node di una lista

Partendo dal primo elemento della lista, ad ogni chiamata in **prt** troviamo l'indirizzo del prossimo elemento della lista fino a quando non viene raggiunta nuovamente la testa.

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

# Process Management

## Process descriptor

La struttura dati **task_struct** è la struttura dati usata dal kernel per rappresentare un processo.

```c
struct task_struct {
    pid_t pid;                    // the process identifier
    unsigned long state;          // the current process's state
    struct task_struct *parent;   // pointer to parent process
    struct list_head children;    // list of children process
    struct list_head siblings;    // list of siblings process
    struct mm_struct *mm;         // memory descriptor
    struct sched_entity se;       // time accounting
    struct thread_struct thread; // context of process
}
```

### Process identifier

Il **PID** è un valore numerico che identifica un processo. Quando si crea un nuovo processo, viene generato un nuovo PID sommando 1 all'ultimo PID.
In Linux, il valore massimo per un PID è **32768**. Quando viene raggiunto il PID massimo, l'ultimo PID assegnato viene resettato a 0 prima di cercare un nuovo PID.
La macro **RESERVED_PID** (solitamente = 300) è definita per riservare dei PIDs ai processi di sistema e daemons (servizi come web server). Tutti i processi utente hanno **PID > RESERVED_PID**.

### Stato di un processo

Lo stato di un processo è un valore numerico che descrive lo stato corrente del processo. Un processo può trovarsi in uno dei seguenti stati:

- ➢ **TASK_RUNNING**: il processo sta eseguendo, o ha tutte le risorse per essere eseguito, eccetto la CPU
- ➢ **TASK_INTERRUPTIBLE**: il processo è bloccato (sleep), aspetta una certa condizione per eseguire. Quando si verifica la condizione, il kernel imposta lo stato del processo a TASK_RUNNING. Il processo può svegliarsi e diventare eseguibile anche se riceve un segnale.
- ➢ **TASK_UNINTERRUPTIBLE**: è molto simile a TASK_INTERRUPTIBLE, eccetto che non può essere risvegliato da un segnale. Viene utilizzato in situazioni in cui il processo deve aspettare senza interruzioni
- ➢ **TASK_STOPPED**: il processo è stoppato, non è in esecuzione e non è idoneo per essere eseguito

- **EXIT_ZOMBIE**: l'esecuzione del processo è terminata, ma il processo padre non ha ancora eseguito una *wait4(0)* o *waitpid()* system call per tornare informazioni sul processo morto.
- **EXIT_DIED**: **stato finale** -> il processo viene rimosso dal sistema perchè il processo padre ha eseguito una *wait4(0)* o *waitpid()* system call

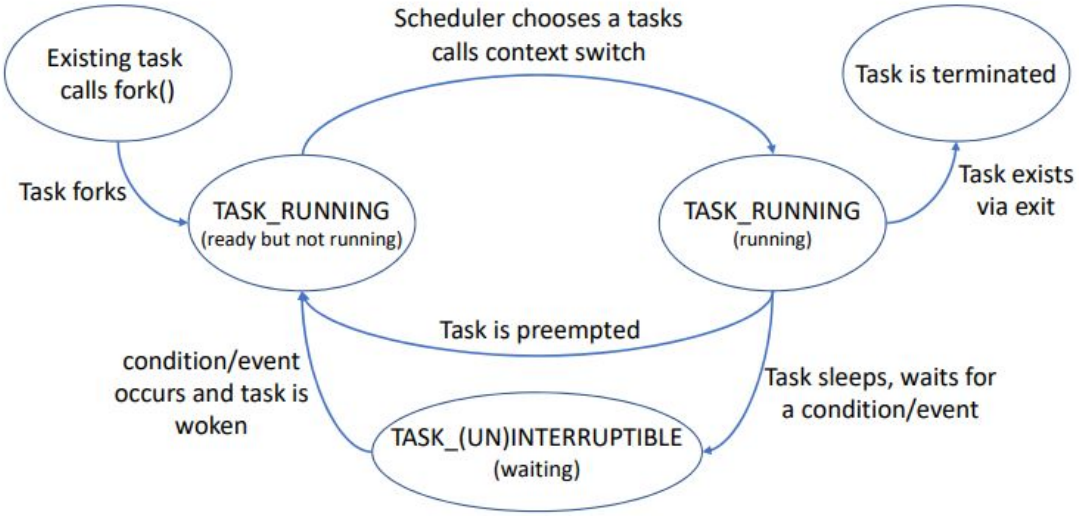
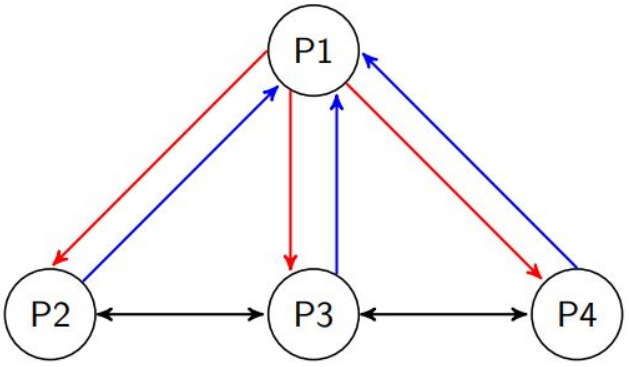



Figure: Flow chart of process states

## Relazioni tra i processi

I processi creati da un programma hanno relazioni con parent/child. Quando un processo crea più figli, questi figli hanno delle relazioni di pari livello (fratelli).
I campi della **task_struct** che descrivono queste relazioni tra processi sono:

- **parent**: puntatore al processo padre
- **child**: head della lista contenente tutti i figli creati dal processo
- **sibling**: head della lista contenente tutti i figli creati dal processo padre




## Time accounting

Il campo **se** di tipo **struct sched_entity** riporta la priorità e i tempi di esecuzione di un processo.

```
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
}
```

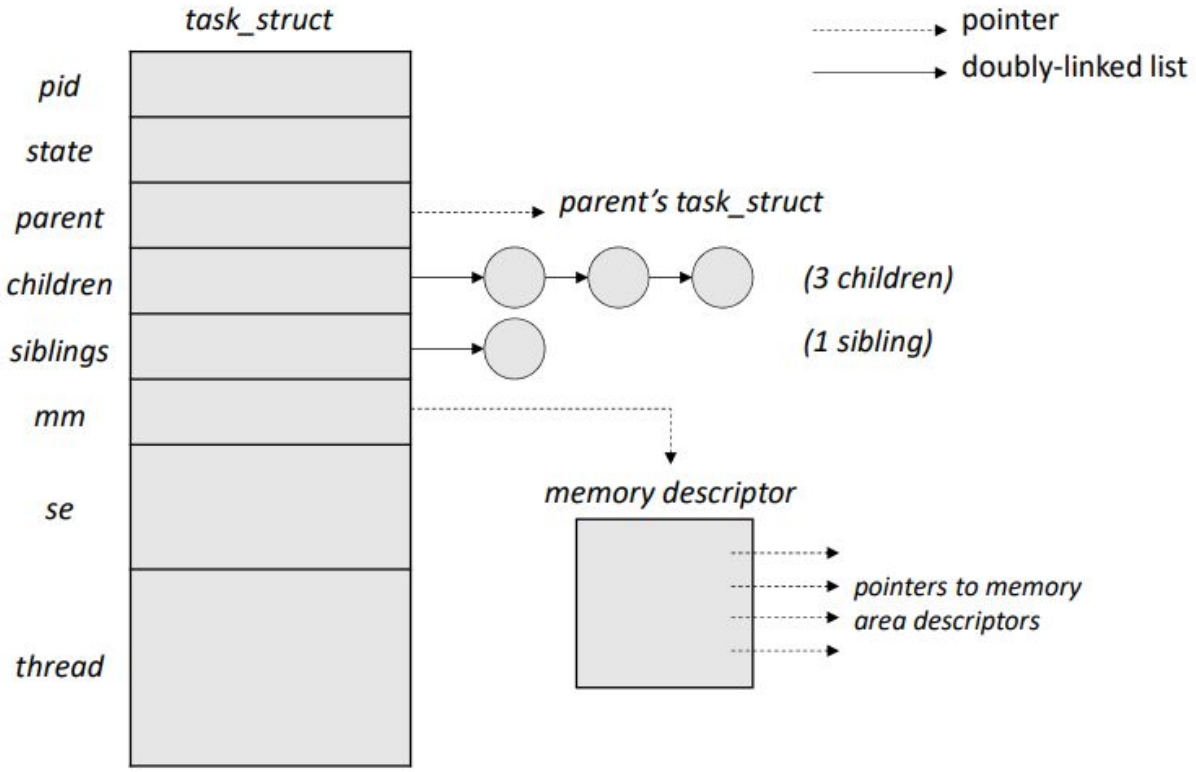
- **prio**: definisce la priorità di esecuzione di un processo. Ha valore nel range [100, 139], con **100** la **massima priorità**. Di default, la priorità di un processo nuovo è **120**. Un processo può incrementare/decrementare il valore di *prio* usando la system call **nice(inc)**, che prende in input un valore nel range [-20,19]
- **start_runtime**: riporta quando il processo è stato eseguito la prima volta
- **exec_start**: riporta quando il proecsso è stato eseguito l'ultima volta
- **sum_exec_runtime**: il tempo di esecuzione complessivo impiegato dal processo della CPU
- **vruntime**: il runtime virtuale, ovvero il tempo di esecuzione complessivo ponderato speso dal processo in CPU

## Contesto di un processo

Il campo **thread** di tipo **struct thread_struct** riporta il contesto di un processo ogni volta che viene tolto dalla CPU.

```
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
}
```

## Rappresentazione in memoria della task_struct




# Scheduler

## Data structures

La struttura dati **runqueue** è la più importante struttura dati dello scheduler. **Contiene tutti i processi del sistema in 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 queue;   // list of processes in running state
}
```

**NOTA**: **queue** è la testa di una circular, doubly-linked list che contiene tutti i processi del sistema in running state. Di conseguenza, un campo **run_list** di tipo struct_list_head è aggiunto alla **struct task_struct**.

## Scheduler execution flow

Lo scheduler è chiamato dopo l'intercettazione di un interrupt/eccezione. In dettaglio, lo scheduler esegue le seguenti operazioni:

- aggiorna le variabili di contabilità temporale del processo corrente
- prova a svegliare uno dei processi in attesa. Se una delle condizioni di attesa è rispettata, un processo viene svegliato impostando il suo stato a running, e viene inserito nella **runqueue**
- l'algoritmo di scheduling seleziona il processo successivo che deve essere eseguito dalla CPU prendendolo dalla runqueue
- esegue il context-switch

## Algoritmi di scheduling

**pick_next_task** è la funzione chiamata dallo scheduler per prendere il prossimo processo da eseguire. A seconda dell'algoritmo di scheduling implementato, il processo successivo può essere scelto differentemente.
MentOs mette a disposizione tre algoritmi (**nota**: per collezionare tutti i processi in running state, si usa una doubly-linked list):

- **RR** (Round Robin)
- **Priority** (Highest Priority First)
- **CFS** (Completely Fair Scheduler)

## Round Robin

Round Robin è un algoritmo in cui viene assegnato un time slice fisso ad ogni processo, in una via ciclica. Semplice, preemptive, facile da implementare, starvation-free.

```
struct task_struct * pick_next_task(struct runqueue *runqueue) {
    // nNode = next(c)
    struct list_head *nNode = runqueue->curr->run_list.next;

    // if isHead(L, nNode)
    if (nNode == &runqueue->queue)
        nNode = nNode->next;

    // n = entry(nNode)
    task_struct *next = list_entry(nNode, struct task_struct, run_list);
    return next;
}
```

L'algoritmo RR assume che tutti i processi sono ugualmente importanti.
Ma non è sempre così, inoltre alcuni utenti potrebbero avere stato differente e i processi di un amministratore per esempio potrebbero essere più prioritari di altri.

## Highest Priority First

Ogni processo ha una **priorità statica**. Più piccolo è il numero, più alta è la priorità del processo. Semplicemente, lo scheduler prende il processo con più alta priorità. Un processo potrebbe essere preempted se un altro processo con priorità più alta arriva nella run queue.
**Vantaggio**: la priorità fornisce un buon meccanismo in cui può essere definita la relativa importanza di ogni processo
**Svantaggio**: se i processi con alta priorità usano molto tempo di CPU, quelli con priorità bassa potrebbero venire posticipati all'infinito -> **starvation**

**Require:** Current process c, List of processes L
**Ensure:** Next process n
1: n = c
2: **for all** INode ∈ L **do**
3: t = entry(INode)
4: **if** priority(t) < priority(n) **then**
5: n = t
6: **end if**
7: **end for**

**Algorithm 2:** pseudocode of Highest Priority First.

## Completely Fair Scheduler

Punta a prevenire la starvation assegnando la CPU equamente a tutti i processi di sistema. **IDEA**: usare la priorità di ogni processo per "ponderare" il suo totale tempo di esecuzione (**virtual runtime**). I processi con priorità bassa hanno un runtime virtuale che aumenta più rapidamente rispetto ai processi con una priorità elevata. **Lo scheduler sceglie sempre il processo con il minor virtual runtime.**

Lo scheduler ha bisogno di conoscere il peso dei task per stimare il suo tempo di CPU. Quindi, il numero di priorità deve essere associato a tale peso:

```
static const int prio_to_weight[] = {
    /* 100 */     88761, 71755, 56483, 46273, 36291,
    /* 105 */     29154, 23254, 18705, 14949, 11916,
    /* 110 */      9548,  7620,  6100,  4904,  3906,
    /* 115 */      3121,  2501,  1991,  1586,  1277,
    /* 120 */      1024,   820,   655,   526,   423,
    /* 125 */       335,   272,   215,   172,   137,
    /* 130 */       110,    87,    70,    56,    45,
    /* 135 */        36,    29,    23,    18,    15
};
```

Notiamo che il rapporto tra due voci successive nell'array è quasi **1.25**.
Questo numero è scelto in modo tale che:

- se la priorità del task viene **ridotta di 1**, **aumenta del 10%** la quota di tempo CPU disponibile
- se la priorità del task viene **aumentata di 1, allora si riduce del 10%** la quota di tempo CPU disponibile

Dato l'array **prio_to_weight** è possibile aggiornare il virtual runtime di un processo p, ovvero la sua esecuzione ponderata usando la formula:

**vruntime += delta_exec * (NICE_0_LOAD / weight(p))**
in cui:

- **vruntime:** il virtual runtime del processo
- **delta_exec**: l'ultimo tempo impiegato da p nella CPU
- **NICE_0_LOAD:** il peso di un task con priorità normale (1024)
- **weight(p)**: il peso di p definito dall'array prio_to_weight

**Require:** Current process c, List of processes L
**Ensure:** Next process n
1: updateVirtualRuntime(c)
2: n = c
3: **for all** INode ∈ L **do**
4: t = entry(INode)
5: **if** virtualRuntime(t) < virtualRuntime(n) **then**
6: n = t
7: **end if**
8: **end for**

**Algorithm 3:** Pseudocode of Completely Fair Scheduler.

## Scheduler - Context switch

Vedi nelle slide mentos_process_management da pag.49

# LEZIONE 7 - Memory management (MentOS)

## Physical Memory Management

In un sistema a 32 bit, lo **spazio di indirizzamento** (4GB) della RAM viene diviso in **page frame**. Sono supportate pagine di grandezza 4KB, 2MB, 4MB. La grandezza tipica è **4KB**.
L'unità base su cui ragionare sono quindi i page frame.

### Page descriptor

Il kernel deve tenere traccia dello stato di ogni page frame. Per ogni istanza, deve poter determinare se: è libera, contiene codice o strutture dati del kernel, appartiene ad un processo in user mode,...
La struttura **struct page_t** contiene le informazioni di stato di un page frame:

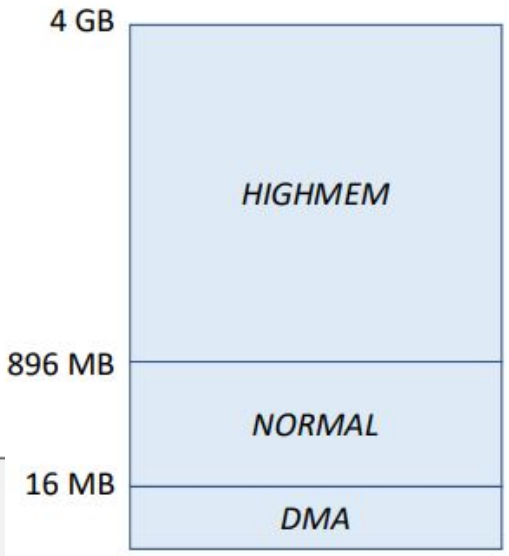
- **_count**: se vale **-1**, significa che il page frame è **libero**. Altrimenti, il page frame è assegnato ad uno o più processi o è usato dal kernel
- **private** (usato dal Buddy System): quando la pagina è libera
- **lru** (usato dal Buddy System): punta all'ultima lista di pagine doppiamente concatenate a cui questa page_t è appartenuta

```
struct page_t {
    int _count;
    unsigned int private;
    struct list_head lru;
    //... continue
}
```

### Zone descriptor

Il kernel partiziona la memoria fisica in tre zone:

- **ZONE_DMA**: contiene tutti i page frame che stanno dai 16MB in giù
- **ZONE_NORMAL**: da 16 a 896MB, contiene una serie di page frame che sono regolarmente mappati
- **ZONE_HIGHMEM**: da 896MB, chiamata anche **memoria alta**. Contiene tutti i page frame che non sono permanentemente mappati




Ogni zona di memoria ha il suo descriptor del tipo di zona.

```
struct zone {
    unsigned long free_pages;          // Number of free pages in the zone.
    free_area_t free_area[MAX_ORDER]; // buddy blocks (see next)
    page_t * zone_mem_map;             // pointer to first page descriptor
    uint32_t zone_start_pfn;           // Index of the first page frame
    unsigned long size;                // Total size of zone in pages
    char * name;                       // Name of the zone
}
```

### Zoned page frame allocator

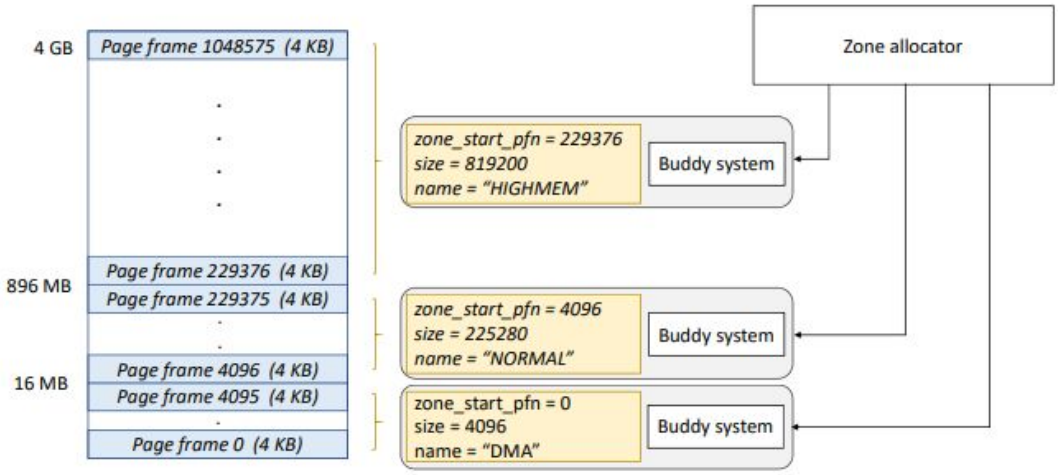



Figure: Zoned page frame allocator

**Zone page allocator** è un sottosistema del kernel per gestire le richieste di allocazione/deallocazione di memoria per gruppi contigui di page frames.
Mette a disposizione due funzioni:

- **alloc_pages(zone, order)**: usata per richiedere $2^{order}$ page frames contigui a partire da una zona. Se c'è la quantità richiesta di allocazione ritorna la prima **page_t** del blocco, altrimenti ritorna NULL se l'allocazione fallisce.
- **free_pages(page, order)**: usata per rilasciare $2^{order}$ page frames contigui di una zona

**NOTA**: usualmente queste funzioni non ricevono la zona come argomento, ma si utilizzano invece delle macro ben definite (es. flag Get Free Page)

## Buddy System

Il Buddy system è un metodo efficiente per allocare gruppi di page frames contigui della potenza di due.
Tutti i page frames liberi sono raggruppati in 11 liste di blocchi che contengono gruppi di 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 e 1024 page frames contigui.
La richiesta più grande di 1024 page frames corrisponde a un blocco di 4MB di RAM contigua.
La richiesta più piccola è un page frame, che corrisponde a un blocco di 4KB di RAM contigua.

private contiene l'ordine a cui il page frame contiene
**(Vedi l'esempio di esecuzione dell'algoritmo nelle slide)**

## PARTE DI ALLOC

Cerca un blocco per soddisfare la richiesta:

**Require:** free_area array f, Request Order ro
**Ensure:** Found Order of a not empty free_area, or NULL

```
1: fo = ro
2: while fo < MAX_ORDER do
3:    if !empty(f[fo]) then
4:       return fo
5:    end if
6:    fo = fo + 1
7: end while
8: return NULL
```

**Algorithm 1:** Search for a block big enough to satisfy the request.

Rimuove il blocco dai page frames liberi:

**Require:** free_area array f, found order of a not empty free_area fo
**Ensure:** A block of page frames

```
1: block = getFirstBlock(f[fo])
2: removeBlock(f[fo], block)
3: return block
```

**Algorithm 2:** Remove a block of free page frames.

Split:

**Require:** free_area array f, Request Order ro, Found Order fo

```
1: while fo > ro do
2:    free_block = splitRight(block)
3:    fo = fo - 1
4:    addBlock(f[fo], free_block)
5:    block = splitLeft(block)
6: end while
```

**Algorithm 3:** Split block until it is just big enough for the request.
The function *splitRight* takes in input a block, and return its right half.
The function *splitLeft* takes in input a block, and return its left half.

## PARTE FREE PAGES

**Require:** free_area array f, Block b, Order o

```
1:  while o < MAX_ORDER - 1 do
2:      buddy = getBuddy(b, o)
3:      if !free(buddy) | order(buddy) ≠ o then
4:          break;
5:      end if
6:      removeBlock(f[o], buddy)
7:      if buddy < b then
8:          b = buddy
9:      end if
10:     o = o + 1
11: end while
12: addBlock(f[o], b)
```

**Algorithm 4:** Search for a block big enough to satisfy the request.

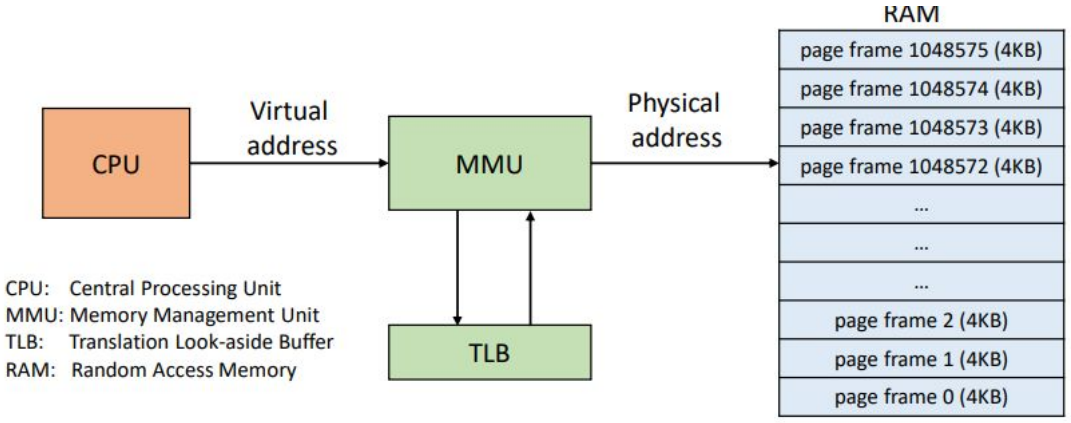
## Virtual Memory Management

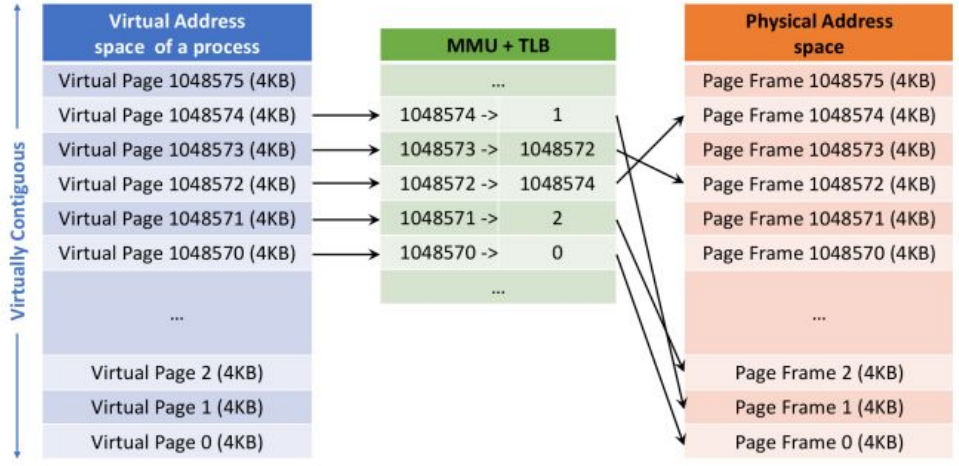
Il kernel utilizza la memoria virtuale per mappare indirizzi virtuali a indirizzi fisici.
**Vantaggi**:

- la ram può essere virtualmente divisa in kernel e user space
- ogni singolo page frame può avere diversi permessi di accesso
- ogni processo ha il suo memory mapping
- un processo può accedere solo ad un sottoinsieme della memoria fisica disponibile
- un processo può essere rilocato

### Memory Management Unit (MMU)

La **MMU** è un **componente hw** che mappa gli indirizzi virtuali in indirizzi fisici. **Vantaggi**: il mapping è fatto da hw -> alte performance.



**DA MEMORIA VIRTUALE A FISICA**
Vedi nelle slide

**Memory descriptor**