SO_LabNotes

Thursday, October 3, 2024 9:10 AM
Class: Operating Systems - Sistemi Operativi
Prof: Claudio Schifanella
Author: Michele Scarlata
Status: #Revisioning
Topics: #Lab #Linux

- LINUX MAN PAGES
- GNU GCC

- MuPIn : Museo Piemontese dell'Informatica

ARGOMENTI DEL LABORATORIO UNIX

1. introduzione a UNIX;
2. integrazione C: operatori bitwise, precedenze, preprocessore, pacchettizzazione del codice, compilazione condizionale e utility make;
3. controllo dei processi;
4. pipe e fifo;
5. code di messaggi;
6. memoria condivisa;
7. semafori;
8. segnali;
9. introduzione alla programmazione bash.

COMPONENTI PRINCIPALI

Un SO è l'interfaccia software tra l'utente e le componenti hardware del sistema.
Un SO di tipo UNIX è composto da:

- Unix Kernel
- Shell
- File System
- Device Drivers
- Utilities (Applicazioni)

Define

- Le macro possono essere dichiarate anche a compile time: `gcc -D PI=3.14` equivale a fare `#define PI 3.14` nel file.
- Spesso si definiscono anche macro senza valore, usate per parametri condizionali:
  `#define DEBUG`
- Vi sono delle macro sostituite da informazioni di sistema e/o del file in compile time:
  1. `__LINE__` A decimal constant representing the current line number.
  2. `__FILE__` A string representing the current name of the source code file.
  3. `__DATE__` A string representing the current date when compiling began for the current source file. It is in the format "mmm dd yyyy", the same as what is generated by the asctime function.
  4. `__TIME__` A string literal representing the current time when compiling began for the current source file. It is in the format "hh:mm:LiBadgeSwissFranc:", the same as what is generated by the asctime function.
- Per le define multi linea si usa `\`:

```
#define SUM (X, Y)\
((X) + (Y))
```

- I parametri di una macro, se indicati con #, vengono sostituiti con la stringa del nome stesso e non con il valore:

```
#define PRINT_INTV(V) printf("%s = %i\n", #V, V);
```

Classi di memorizzazione

Tutte le variabili in C, oltre al tipo, hanno una delle 4 seguenti classi di memorizzazione:

- auto
- extern
- register
- static

auto

Quella di default, può essere omessa. Le variabili di questo tipo sono considerate locali al blocco in cui sono dichiarate, all'uscita dal blocco il sistema considera tale memoria come libera, causando la perdita dei loro valori. Se, eventualmente si fa ritorno al blocco, i valori precedenti sono ormai andati persi.

> ⚠ **non vengono inizializzate dal sistema, e quindi contengono inizialmente valori «sporchi»**

extern

Sono variabili definite ed allocate in altri file (un altro programma, il sistema operativo...). Anche le funzioni possono essere definite extern. Il compilatore assume che questa variabile esiste (e non alloca dunque spazio per essa) tuttavia, se in linking time tale variabile non è effettivamente presente, viene generato un errore.

register

Ha come obbiettivo l'aumento della velocità di esecuzione, segnalando al compilatore che tale variabile dovrebbe essere memorizzata in registri di memoria ad alta velocità, qualora il compilatore non possa allocare un registro fisico, viene utilizzata la classe `auto`.
quando la velocità è importante, il programmatore può scegliere poche variabili alle quali viene fatto più frequentemente accesso.
la variabile register è di norma dichiarata nel punto più vicino possibile al punto in cui viene utilizzata, per consentire la massima disponibilità di registri fisici, utilizzati solo quando necessario.

> ⚠ **non vengono inizializzate dal sistema, e quindi contengono inizialmente valori «sporchi»**

static

servono per permettere a una variabile locale di mantenere il valore precedente al rientro in un blocco. Sono inizializzate a zero dal sistema, anche array, stringhe, puntatori, strutture e union subiscono lo stesso trattamento: per array e stringhe ogni elemento viene posto a zero.

static (extern)

È possibile definire variabili `static` anche all'esterno di un blocco funzione, rendendole accessibili solamente dal resto del file al di sotto della loro dichiarazione, fungendo dunque da variabile globale comune solo ad una famiglia di funzioni, e solamente all'interno di quel singolo file.

Stack VS Heap VS BSS

Stack
Le variabili dichiarate all'interno di una funzione (compresi i parametri), sono allocate nello stack all'inizio della funzione, modificando lo stack pointer, e deallocate quando la funzione termina.

Heap
Hanno un allocazione dinamica decisa a runtime. Lo heap è utilizzato quando uno spazio di memoria per una variabile è richiesto attraverso le primitive quali:

```
void* malloc (size_t size);
void* calloc (size_t nmemb, size_t size);
void* realloc (void *ptr, size_t size);
```

va poi deallocata tale memoria:

```
void free (void *ptr);
```

BSS (Block Started by Symbol)
La BSS è una parte di memoria di un programma che contiene variabili `global` e `static` non inizializzate. È una delle zone di memoria chiave di un eseguibile.

Viene allocato ed inizializzato a zero a runtime, ed è localizzato tra la sezione `data` (variabili `global` e `static` inizializzate) e lo `heap`.

```
#include <stdio.h>

int global_var;       // Uninitialized global variable → Goes to
BSS
static int static_var; // Uninitialized static variable → Goes to
BSS

int main() {
    printf("Global: %d, Static: %d\n", global_var, static_var);
    return 0;
}
```

🔥 BSS vs. Other Segments

Segment	Stores	Initialized?	Allocated?
Text	Code/Instructions	Yes	Yes
Data	Global/Static (initialized)	Yes	Yes
BSS	Global/Static (uninitialized)	No (but zeroed)	Yes
Heap	Dynamically allocated (malloc, new)	No	Yes
Stack	Local variables, function calls	No	Yes

Compile and link

- `gcc -c filename.c` Crea il file oggetto
- `gcc filename1.o filename2.o -o eseguibile` Linka i vari file oggetto in un unico eseguibile

Variabili d'ambiente

- Ogni processo ha associato un array di stringhe che contiene l'elenco delle variabili di ambiente
- Ogni elemento dell'array ha la forma nome=valore (esempi: HOME, LOGNAME, PATH, ...)
- Ogni volta che un processo viene creato, eredita le variabili di ambiente del processo creatore

- All'interno di un programma C, la lista delle variabili di ambiente può essere consultato usando la variabile globale
  - `char **environ`
  - La variabile environ è un NULL-terminated array
  La funzione

```
char * getenv (const char *name)
```

restituisce la stringa della variabile di ambiente passata come parametro

Utility make

Si basa sull'utilizzo di un file nominato `makefile` che descrive le dipendenze presenti nel progetto.

- 👍 Favorisce le ricompilazioni frequenti di grandi progetti
- 👎 Richiede tempo per essere scritto correttamente
  Tolti progetti davvero piccoli, spesso viene considerato favorevole fare utilizzo di un `makefile`.

PDF 3 - CONTROLLO PROCESSI

⚠ **Necessitano della libreria UNIX <unistd.h> e <sys/types.h>**

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

PID

Ogni processo ha un ID associato (PID), un intero positivo che identifica univocamente il processo nel sistema.

```
pid_t getpid(void);
// Restituisce il PID (Process IDentifier) del processo
chiamante.
```

Ogni processo ha un genitore: il processo che lo ha creato. La system call getppid() permette ad un processo di conoscere il process ID del proprio padre. Le relazioni tra processi costituiscono una struttura ad albero. Il genitore di ogni

processo ha a sua volta un genitore, fino ad arrivare alla radice: il processo `init`.

```
pid_t getppid(void);
// Restituisce il PID  del padre del processo chiamante.
```

Stato corrente di un processo

ps -aux

```
ps -aux
```

Restituisce l'elenco dei processi in esecuzione rispetto all'utente attuale.

```
ps -aux | grep PID
```

Identifica e stampa su terminale il processo col PID cercato.

ps -la

```
ps -la
```

Select all processes except both session leaders (see getsid(2)) and processes not associated with a terminal.

ps -axjf

```
ps -axjf
```

Restituisce dettagli riguardanti l'alberatura dei processi.

top

```
top
```

interfaccia che contiene tutti i processi in esecuzione sulla macchina, costantemente aggiornata.

htop

```
htop
```

interfaccia che contiene tutti i processi in esecuzione sulla macchina, costantemente aggiornata. L'interfaccia del terminale emula il task manager di Windows.
(Se non presente come programma, va scaricato: `sudo apt-get install htop`).

ps -aux | grep PROCESS_NAME | wc -l

```
ps -aux | grep PROCESS_NAME | wc -l
```

Restituisce il numero+1 dei processi "PROCESS_NAME" in esecuzione.

> *(Es: restituisce 12 -> in realtà sono 11)*

Osserviamo in realtà che per il funzionamento di `grep`, al posto di `PROCESS_NAME` posso usare il `PID`.

cat /proc/PID/status

```
cat /proc/2166/status
```

È anche possibile usare il file system `/proc` per determinare il numero di cambi di contesto per un determinato processo. Il contenuto del file `/proc/2166/status`, per esempio, offre varie statistiche relative al processo con pid = 2166.

> *tra le statistiche: il numero di cambi di contesto avvenuti durante l'esistenza del processo, distinguendo tra cambi di contesto volontari e involontari.*

Vedi: CONTEXT SWITCH

STAT Code	Description
S	Sleeping. Usually waiting for an event to occur, such as a signal or input to become available.
R	Running. Strictly speaking, "runnable," that is, on the run queue either executing or about to run.
D	Uninterruptible Sleep (Waiting). Usually waiting for input or output to complete.
T	Stopped. Usually stopped by shell job control or the process is under the control of a debugger.
Z	Defunct or "zombie" process.
N	Low priority task, "nice."
W	Paging. (Not for Linux kernel 2.6 onwards.)
s	Process is a session leader.
+	Process is in the foreground process group.
l	Process is multithreaded.
<	High priority task.

Controllo dei processi

Con controllo dei processi si indica un insieme di operazioni che include la creazione di nuovi processi, l'esecuzione di processi, e la loro terminazione. A queste operazioni corrispondono le system call `fork()`, `exit()`, `wait()`, `waitpid()` e `execve()`.

---

fork()

```
fork();
```

Crea un nuovo processo figlio come copia QUASI esatta del processo padre:

- ottiene copie di stack, data, heap, and text segments del padre.
- Il termine fork deriva dal fatto che ci si può raffigurare il processo padre come un processo che si suddivide in due

Restituisce:

- `-1` quando il SO non è in grado di generare un nuovo processo.
- Al padre: il PID del processo figlio generato.
- Al figlio: 0
  Nello spazio di indirizzamento del processo figlio viene creata una copia

delle variabili del padre, col valore loro assegnato al momento della fork. Il nuovo processo incomincia l'esecuzione a partire dalla prima istruzione successiva alla fork che lo ha creato.

I due processi eseguono lo stesso testo, ma mantenendo **copie distinte** di stack, data, heap, buffer I/O.

> *stack, dati, heap, buffer I/O del figlio sono inizialmente esatti duplicati delle corrispondenti parti della memoria del padre.*

Dopo la `fork()`, ogni processo può modificare le variabili in tali segmenti senza influenzare l'altro processo.

esempio di utilizzo:

```
pid_t f = fork();

if (f == -1) {exit(1)}; // In caso di errore
if (f) { // Se sono il padre, e dunque f è il pid di mio figlio
//...
} else { // Se sono il figlio
//...
}
```

oppure (preferibilmente):

```
pid_t childpid;
switch (childpid = fork()) {
case -1:
    // Error handling
case 0:
    // Child actions
default:
    // Parent actions
}
```

Quale dei due processi sarà scelto con priorità sulla CPU immediatamente dopo la `fork()`?

Dopo una fork(), è indeterminato quale dei due processi sarà scelto per ottenere la CPU. In programmi scritti male, questa indeterminatezza può

causare errori.
Se invece abbiamo bisogno di garantire un particolare ordine di esecuzione, è necessario utilizzare una qualche tecnica di sincronizzazione (ad esempio: i semafori)

File sharing tra padre e figlio

All'esecuzione della `fork()`, il figlio riceve duplicati di tutti i descrittori di file del padre. Quindi, gli attributi di un file aperto sono condivisi fra genitore e figlio. Per esempio, se il figlio aggiorna l'offset del file, tale modifica è visibile attraverso il corrispondente descrittore nel padre.

Terminazione di un processo

Quando un processo termina:

- Le funzioni che sono state registrate come handler con la funzione `atexit()` vengono eseguite in ordine inverso rispetto alla registrazione
- Tutti gli stream, come ad esempio i files, sono svuotati e chiusi
- Un segnale di `SIGCHILD` è mandato al processo padre
- Ogni figlio del processo appena terminato è assegnato ad un nuovo processo padre, il processo `init`
- Le risorse (memoria, descrittori di file aperti) sono rilasciate
- Il valore di stato in uscita viene memorizzato (troncato a 8 bit)

Un processo può essere terminato da:

- La system call `exit(status)`
- La ricezione di un segnale, ad esempio inviato premendo CTRL+C. Nel caso succeda questo, le funzioni registrate con `atexit` non sono invocate

Il valore di stato di ritorno può dare le informazioni sul motivo della terminazione del processo. Il suo valore può essere compreso tra 0 e 255. Due macro sono definite in `stdlib.h` (e dovrebbero essere usate):

- `EXIT_SUCCESS` (di solito 0)
- `EXIT_FAILURE` (di solito 1)

_exit()

Un processo può terminare in due modi principali.

1. Una di queste è la terminazione inaspettata, causata dalla ricezione di un segnale, la cui azione di default è di terminare il processo (con o senza la generazione di un file di dump)
2. In alternativa, un processo può terminare normalmente usando la system call `_exit()`

```
#include <unistd.h>
void _exit(int status);
```

L'argomento status passato alla funzione `_exit()` definisce lo stato di terminazione del processo, che sarà disponibile al processo padre quando egli chiamerà la funzione `wait()`. Sebbene sia definito come un intero, solamente gli otto bit più significativi sono usati. Per convenzione, un valore di terminazione di 0 indica che il processo è terminato con successo, mentre un valore diverso da 0 una una terminazione inaspettata.

exit(status)

Di solito, il codice dei programmi non contengono la chiamata alla funzione `_exit()`, ma contengono invece la chiamata a `exit()`.
Le azioni eseguite dalla exit() sono:

- Gli exit handlers (le funzioni registrate con `atexit()` e `on_exit()`) sono invocate;
- I buffer dello stream `stdio` stream viene svuotato
- La funzione `_exit()` viene invocata, usando il valore di stato passato come parametro.

```
void exit(int status);
```

- La funzione di libreria `exit(status)` termina un processo, rendendo le risorse utilizzate dal processo (memoria, descrittori dei file aperti, etc.) nuovamente disponibili per essere allocate dal kernel.
- l'argomento `status` è un intero che descrive lo stato di terminazione del processo: utilizzando la system call `wait()` il processo padre può risalire a

tale `status`.

Monitoraggio dei processi

wait()

```
#include <sys/wait.h>
pid_t wait(int *status);
```

In molti casi un processo padre deve essere informato quando uno dei figli cambia stato, quando termina o è bloccato da un segnale.
Con la system call `wait()` un processo genitore attende che uno dei processi figli termini e scrive lo stato di terminazione di quel figlio nel buffer puntato da status.
`wait()` restituisce:

- `-1` in caso di errore
- `PID` del processo figlio terminato

Se nessun figlio del processo chiamante ha già terminato, la chiamata si blocca finché uno dei figli termina; se un figlio ha già terminato al momento della chiamata, `wait()` restituisce immediatamente.
Se status non è `NULL`, l'informazione sulla terminazione del figlio è assegnata all'intero cui punta status (NB: status è di tipo int*).

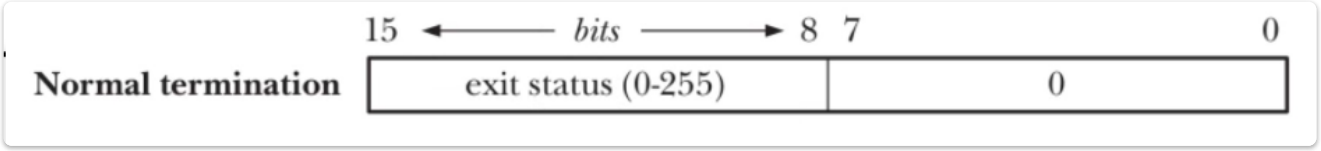
In caso si sia verificato un errore: un possibile errore è che il processo chiamante potrebbe non avere figli, il che è indicato dal valore `ECHILD` di `errno`.
Possiamo quindi utilizzare il seguente ciclo per attendere la terminazione di tutti i figli di un processo:

```
while ((childPid = wait(NULL)) != -1)
    continue;
if (errno != ECHILD) // errore inatteso
    errExit("wait"); // gestione errore...
```

> *il while continua finché la `wait()` può ancora attendere la terminazione di almeno uno dei figli, dopo il while, dal quale esco solo se `wait()` mi restituisce -1 (errore), controllo `errno` per accertarmi che l'errore sia stato causato dall'assenza di figli da poter attendere.*

Il parametro `status` viene usato per comunicare al processo padre lo stato di terminazione del processo figlio (se terminato correttamente).
Viene passato il valore indicato nella funzione `exit()` del figlio (ma con una particolare codifica...).

La macro `WEXITSTATUS` (status) viene utilizzata per estrarre il valore di ritorno del processo figlio.




```c
/*
Shift a destra di 8 bit, mi permette di prendere gli 8 bit più significativi
*/
#define WEXITSTATUS(X) ((X) >> 8)

int status=0;
while ((child_pid = wait(&status)) != -1) {
    printf("Valore di ritorno: %d\n", WEXITSTATUS(status));
}
```

---

waitpid()

⚠ Problemi della wait():

1. Se un processo padre ha creato vari figli, non è possibile attendere la terminazione di un particolare figlio con la `wait()`; la `wait()` permette semplicemente di attendere che uno dei figli del chiamante termini.
2. Se nessun figlio ha già terminato, la `wait()` si blocca. In alcuni casi è preferibile eseguire una *nonblocking* wait, in modo da ottenere

immediatamente l'informazione che nessun figlio ha ancora terminato la propria esecuzione.

3. Con la `wait()` è possibile avere informazioni sui figli che hanno terminato. Non è invece possibile ricevere notifiche quando un figlio è bloccato da un segnale (come `SIGSTOP`) o quando un figlio stopped è risvegliato da un segnale `SIGCONT`.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
//Returns process ID of child, 0, or –1 on error
```

L'header file `<sys/wait.h>` definisce un insieme standard di macro che possono essere utilizzate per interpretare un *wait status*.

L'argomento pid permette di selezionare il figlio da aspettare, secondo queste regole:

- se `pid` > 0, attendi per il figlio con quel pid.
- se `pid` == 0, attendi per qualsiasi figlio nello stesso gruppo di processi del chiamante (padre).
- se `pid` < -1, attendi per qualsiasi figlio il cui process group è uguale al valore assoluto di pid.
- se `pid` == -1, attendi per un figlio qualsiasi.

> **La chiamata `wait(&status)` è equivalente a `waitpid(–1, &status, 0)`.**

L'argomento options è una bit mask che può includere (in OR) zero o più dei seguenti flag:

- `WUNTRACED`: oltre a restituire info quando un figlio termina, restituisci informazioni quando il figlio viene bloccato da un segnale.
- `WCONTINUED`: restituisci informazioni anche nel caso il figlio sia stopped e venga risvegliato da un segnale `SIGCONT`.
- `WNOHANG`: se nessun figlio specificato da `pid` ha cambiato stato, restituisci immediatamente, invece di bloccare il chiamante. In questo caso, il valore

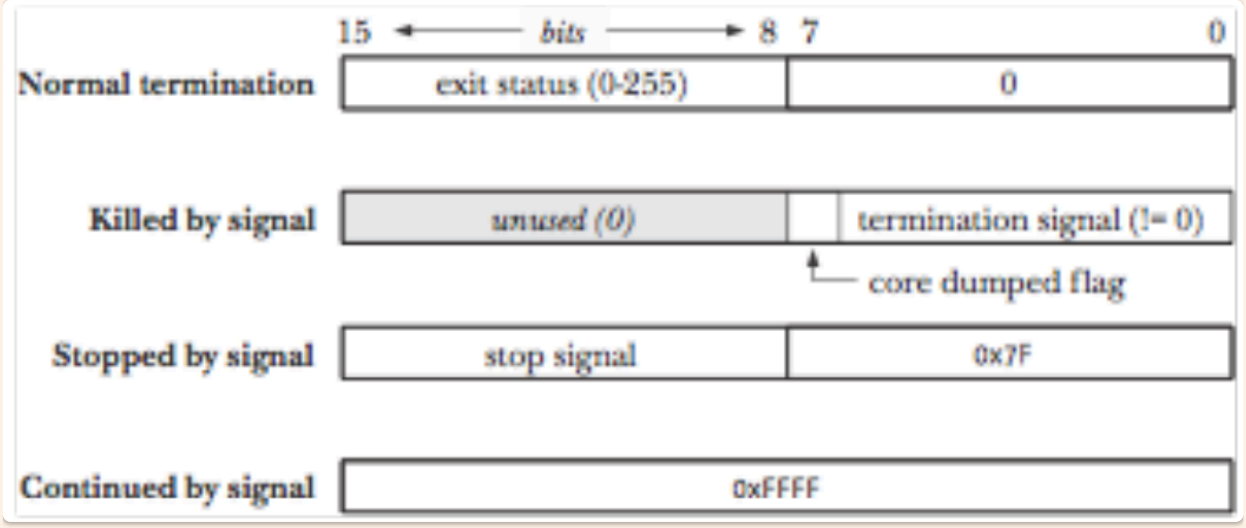
di ritorno di `waitpid()` è 0. Se il processo chiamante non ha figli con il `pid` richiesto, `waitpid()` fallisce con l'errore `ECHILD`.

Il valore status restituito da `wait()` e `waitpid()` ci consente di distinguere fra i seguenti eventi per il figlio:

- Normal termination: il figlio ha terminato l'esecuzione chiamando `_exit()` (o `exit()`), specificando un codice d'uscita (exit status) intero.
- Killed by signal: il figlio ha terminato l'esecuzione per la ricezione di un segnale non gestito.
- Stopped by signal: il figlio è stato bloccato da un segnale, e `waitpid()` è stata chiamata con il flag `WUNTRACED`.
- Continued by signal: Il figlio ha ripreso l'esecuzione per un segnale `SIGCONT`, e `waitpid()` è stata chiamata con il flag `WCONTINUED`.

⚠ Sebbene sia definito come int, solo gli ultimi 2 byte del valore puntato da status sono effettivamente utilizzati.

Il modo in cui questi 2 byte sono scritti dipende da quale è evento è occorso per il figlio:



L'header file `<sys/wait.h>` definisce un insieme standard di macro che possono essere utilizzate per interpretare un wait status.

? Quali sono le differenze tra le macro assunte da `status`?

Applicate allo status restituito da `wait()` o `waitpid()`, solo una delle seguenti macro restituirà true.

- `WIFEXITED(status)`. Restituisce true se il processo figlio è terminato normalmente. In questo caso la macro `WEXITSTATUS(status)` restituisce l'exit status del processo figlio.
- `WIFSIGNALED(status)`. Restituisce true se il figlio è stato ucciso da un segnale. In questo caso, la macro `WTERMSIG(status)` restituisce il numero del segnale che ha causato la terminazione del processo.
- `WIFSTOPPED(status)`. Restituisce true se il figlio è stato bloccato da un segnale. In questo caso, la macro `WSTOPSIG(status)` restituisce il numero del segnale che ha bloccato il processo.
- `WIFCONTINUED(status)`. Restituisce true se il figlio è stato risvegliato da un segnale `SIGCONT`.

Esempio di utilizzo di `waitpid()`:

```
pid_t childPid = waitpid(-1, &status, WUNTRACED | WCONTINUED)
```

`-1`: Attendo, per un figlio qualsiasi, la sua terminazione;
`WUNTRACED`: o che venga *bloccato* da un segnale;
`WCONTINUED`: o che venga *sbloccato* da un segnale `SIGCONT`.

Processi orfani

In generale o il padre sopravvive al figlio, o viceversa.

- il figlio orfano è adottato da `init`, il progenitore di tutti i processi, il cui process ID è 1. In altre parole, dopo che il genitore di un processo figlio termina, una chiamata a `getppid()` restituirà il valore 1 (l'id dipende dal SO). Può essere utile per capire se il vero padre di un processo figlio è ancora vivo.

Processi Zombie

Cosa capita a un figlio che termina prima che il padre abbia avuto modo di eseguire una `wait()`? • Sebbene il figlio abbia terminato, il padre dovrebbe poter avere la possibilità di eseguire una `wait()` in un momento successivo per determinare come è terminato il figlio.
Il kernel garantisce questa possibilità trasformando il figlio in uno zombie. Gran

parte delle risorse gestite da un figlio sono rilasciate al sistema per essere assegnate ad altri processi. L'unica parte del processo che resta è un'entry nella tabella dei processi che registra il process ID del figlio, il termination status, e le statistiche sull'utilizzo delle risorse.

Un processo zombie non può essere ucciso da un segnale, neppure `SIGKILL`. Questo assicura che il genitore possa sempre eventualmente eseguire una `wait()`. *Quando il padre esegue una `wait()`, il kernel rimuove lo zombie*, dal momento che l'ultima informazione sul figlio è stata fornita all'interessato. Se il genitore termina senza fare la `wait()`, il processo `init` adotta il figlio ed esegue automaticamente una `wait()`, rimuovendo dal sistema il processo zombie.

Se un genitore crea un figlio, ma fallisce la relativa `wait()`, un elemento relativo allo zombie sarà mantenuto indefinitamente nella tabella dei processi del kernel. Se il numero degli zombie cresce eccessivamente, gli zombie possono riempire la tabella dei processi, e questo impedirebbe la creazione di altri processi. Poiché gli zombie non possono essere uccisi da un segnale, l'unico modo per rimuoverli dal sistema è uccidere il loro padre (o attendere la sua terminazione). A quel momento gli zombi possono essere adottati da `init` e rimossi.

---

strerror()

The `strerror()` function returns a pointer to a string that describes the error code passed in the argument `errnum`.

```
//errno viene modificato in base all'ultima funzione ad aver
fallito l'esecuzione
if (errno == ECHILD) { //errno == ECHILD : No child processes
  printf("In PID = %d, no more child processes\n", getpid());
  exit(EXIT_SUCCESS);
} else {
  fprintf(stderr, "Error #%d: %s\n", errno, strerror(errno));
  exit(EXIT_FAILURE);
}
```

Unix Signals

> 🖊 **See the full note here:** Unix Signals.

Esecuzione di Programmi

execve(pathname, argv, envp)

- La system call `execve(pathname, argv, envp)` carica un nuovo programma (`pathname`, con il relativo argomento list `argv`, e l'environment `envp`) nella memoria del processo.
- Il testo del programma precedente è cancellato e stack, dati, e heap sono sostituiti dal nuovo programma, per poi iniziare l'esecuzione dalla nuova `main`. Questa operazione è riferita come "execing" di un nuovo programma.
- Varie funzioni di libreria utilizzano la syscall `execve()`, della quale ciascuna costituisce una variazione nell'interfaccia.

> 🔥 **Il padre può comunque sfruttare `waitpid()`**
>
> Quando un processo esegue `execve`, il **contenuto della memoria viene sostituito**, ma il **PID rimane invariato**. Questo perché `execve` non crea un nuovo processo, ma sostituisce l'immagine del processo chiamante con un nuovo programma.

```c
#include <unistd.h>
int execve(const char *pathname, char *const argv[], char *const
envp[])
// Never returns on success, returns -1 on error
```

- L'argomento `pathname` contiene il pathname del programma che sarà caricato nella memoria del processo.
- L'argomento `argv` specifica gli argomenti della linea di comando da passare al nuovo programma. Si tratta di una lista di puntatori a stringa, terminati da puntatore a NULL. Il valore fornito per `argv[0]` corrisponde al nome del comando. Tipicamente, questo valore è lo stesso del basename (i.e., l'ultimo elemento) del pathname.

- L'ultimo argomento, `envp`, specifica la lista environment list per il nuovo programma. L'argomento `envp` corrisponde all'array `environ`; è una lista di puntatori a stringhe (terminata da ptr a NULL) nella forma name=value.

Poiché sostituisce il programma che la ha chiamata, una chiamata di `execve()` che va a buon fine non restituisce. Non abbiamo quindi bisogno di controllare il valore di ritorno di `execve()`; sarà sempre -1. Il fatto che abbia restituito un qualche valore ci informa che è occorso un errore, e come sempre è possibile utilizzare `errno` per determinarne la causa.

Condizioni di errore di execve

Fra gli errori che possono essere restituiti in `errno`:

- `EACCES`. l'argomento pathname non si riferisce a un file normale, il file non è un eseguibile, o una delle componenti del pathname non è ricercabile (i.e., sono negati i permessi di esecuzione sulla directory).
- `ENOENT`. Il file riferito dal pathname non esiste.
- `ENOEXEC`. Il file riferito dal pathname è marcato come un eseguibile ma non è riconosciuto come in un formato effettivamente eseguibile.
- `ETXTBSY`. Il file riferito dal pathname è aperto in scrittura da un altro processo.
- `E2BIG`. Lo spazio complessivo richiesto dalla lista degli argomenti e dalla lista dell'ambiente supera la massima dimensione consentita.

Alternative ad execve()

```
#include <unistd.h>
int execle(const char *pathname, const char *arg, ...
            /* , (char *) NULL, char *const envp[] */);
int execlp(const char *filename, const char *arg, ...
            /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
            /* , (char *) NULL */);

//None of the above returns on success; all return -1 on error
```

Gran parte delle funzioni `exec()` si aspettano un pathname per specificare il nuovo programma da caricare. Invece, `execlp()` e `execvp()` ci consentono di specificare solo il filename. Il filename è cercato nella lista di directory

specificata dalla variabile d'ambiente `PATH`. La variabile d'ambiente `PATH` non è usata se il filename contiene uno slash (/), nel qual caso è trattato come un percorso relativo o assoluto.

> ✎ **Uno degli esempi di lab mostra l'utilità di `execlp` e `execvp`.**

Invece di utilizzare un array per specificare la lista `argv` per il nuovo programma, `execle()`, `execlp()`, e `execl()` richiedono al programmatore di specificare gli argomenti come una lista di stringhe.
La lista di argomenti deve essere terminata da un puntatore a NULL come terminatore della lista. Questo formato è indicato dal `(char *) NULL` commentato nei prototipi riportati sopra. I nomi delle funzioni che richiedono la lista di argomenti come un array (`execve()`, `execvp()`, and `execv()`) contengono la lettera v (da vector).

Le funzioni `execle()` e `execve()` permettono al programmatore di specificare esplicitamente l'environment per il nuovo programma, utilizzando `envp`, un array di puntatori a stringhe terminato dal puntatore a `NULL`. I nomi di queste funzioni terminano con la lettera e (da environment) per indicare questa caratteristica.

File descriptors e exec()

Per default, tutti i descrittori di file aperti da un programma che chiama `exec()` restano aperti attraverso la `exec()` e sono pertanto disponibili per il nuovo programma. Questo è spesso utile, poiché il programma chiamante può aprire file con particolari descrittori, e questi file sono automaticamente disponibili per il nuovo programma, senza che questo debba sapere i loro nomi e/o aprirli.

Eseguire un comando con system()

```c
#include <stdlib.h>
int system(const char *command);
```

La funzione `system()` permette di chiamare un programma per eseguire un comando di shell arbitrario. La funzione `system()` crea un processo figlio che invoca una shell per eseguire il comando `command`.

Esempio di chiamata di system():

```
system("ls -lt | wc -l");
```

Ad esempio, possiamo sfruttare la cosa per far compilare al programma gli eseguibili che deve chiamare ma di cui dispone solo del formato `.c`.

Valore di ritorno di system()

- se command è un NULL pointer, `system()` restituisce un valore diverso da 0 se una shell è disponibile, e 0 se nessuna shell è disponibile.
- se non è stato possibile creare un processo figlio o il suo stato di terminazione non è stato ricevuto, `system()` restituisce -1.
- se non è stato possibile eseguire la shell nel processo figlio, `system()` restituisce un valore come se la shell del processo figlio avesse terminato con la chiamata `_exit(127)`. se tutte le system calls hanno avuto successo, `system()` restituisce lo stato di terminazione della shell figlia utilizzata per eseguire il comando: lo status di terminazione di una shell è lo status di terminazione dell'ultimo comando eseguito.

Vantaggi e svantaggi di system()

Il vantaggio principale offerto da `system()` è la semplicità d'uso:

- non dobbiamo gestire i dettagli relativi alle chiamate di `fork()`, `exec()`, `wait()`, e `exit()`.
- la gestione degli errori e dei segnali è affidata a `system()` per nostro conto.
- poiché `system()` utilizza la shell per eseguire un comando, il processamento legato alle sostituzioni, ridirezioni è effettuato sul comando prima che esso sia eseguito.

Il principale costo della `system()` è l'inefficienza.

- Eseguire un comando usando `system()` richiede la creazione di almeno 2 processi (uno per la shell e uno o più per i comandi eseguiti), ciascuno dei quali esegue una `exec()`.
- Se l'efficienza o la velocità sono richiesti, è preferibile utilizzare chiamate `fork()` e `exec()` per eseguire il programma desiderato.

I segnali

Un segnale è una notifica a un processo che è occorso un certo evento.
Fra i tipi di eventi che causano il fatto che il kernel generi un segnale per un processo ci sono i seguenti:

- È occorsa una eccezione hardware, l'HW ha verificato una condizione di errore che è stata notificata al kernel, il quale a propria volta ha inviato un segnale corrispondente al processo in questione.
  - per esempio, l'esecuzione di istruzioni di linguaggio macchina malformate, divisioni per 0, o riferimenti a parti di memoria inaccessibili.
- L'utente ha digitato sul terminale dei caratteri speciali che generano i segnali.
  - questi caratteri includono il carattere interrupt (normalmente associato a Control-C) e il suspend carattere (Control-Z).
- È occorso un evento software.
  - per esempio, l'input è divenuto disponibile su un descrittore di file, un timer è arrivato a 0, il tempo di processore per il processo è stato superato, o un figlio del processo è terminato.

Nomi simbolici e codici

I segnali sono definiti con interi unici, la cui sequenza inizia da 1. Tali interi sono definiti in `<signal.h>` (o in `<sys/signal.h>`) con nomi simbolici della forma `SIGxxxx`.
Poiché gli effettivi numeri utilizzati per ogni segnale variano a seconda delle implementazioni, all'interno dei programmi è meglio utilizzare questi nomi.
per esempio, quando l'utente digita il carattere dell'interrupt, SIGINT (il segnale numero 2) è inviato a un processo.

> 🔥 **Per approfondimento si signal e i vari nomi simbolici per i segnali:**
>
> `man 7 signal`.

	Function	Character
SIGINT	Terminates the current job.	CTRL + C

	Function	Character
SIGQUIT	Terminates the current job. Makes a core file.	CTRL + \
SIGSTOP	Stops the current job.	CTRL + Z

Ciclo di vita dei segnali

Dopo essere stato generato da un evento, un segnale viene *inviato* (*delivered*) ad un processo, che quindi esegue una qualche azione in risposta al segnale. Fra il momento in cui è generato e il momento in cui è inviato al processo, il segnale si dice *pendente* (*pending*).

Di norma, un segnale pending è inviato ad un processo appena il processo e scelto per l'esecuzione, oppure immediatamente se il processo è già in esecuzione (per esempio, nel caso in cui il processo invii un segnale a se stesso). A volte invece è necessario assicurare che un segmento di codice non sia interrotto dalla consegna di un segnale. In questo caso, possiamo aggiungere un segnale alla *maschera dei segnali del processo*, cioè un insieme di segnali la cui ricezione è attualmente bloccata. Se un segnale è generato mentre il processo è bloccato, il segnale rimane pendente fino a quando non viene successivamente sbloccato e rimosso dalla maschera dei segnali. Varie system call permettono ai processi di aggiungere e rimuovere segnali dalla propria maschera dei segnali.

Ricezione dei segnali

Al momento della ricezione di un segnale, un processo continua con una delle seguenti azioni di default, a seconda del segnale:

- Il segnale è *ignorato*; in questo caso viene scartato dal kernel e non ha effetti sul processo (il processo non è neppure informato del fatto che quel segnale è occorso).
- Il processo viene *terminato* (killed). Questa terminazione è detta anche abnormal process termination, opposta alla terminazione normale del processo, che occorre quando un processo termina usando `exit()`.
- È generato un file contenente un *core dump* file, e il processo viene *terminato*. Un file con core dump contiene un'immagine della memoria virtuale del processo; tale immagine può essere caricata in un debugger per ispezionare lo stato del processo al momento della terminazione.
- Il processo viene *bloccato* (stopped): l'esecuzione è in questo caso sospesa.

- L'esecuzione del processo è *ripresa* (resumed) dopo essere stata bloccata in precedenza.

Unix Trap

Una classe di segnali sono le *trap*: segnali generati da eventi prodotti da un processo ed inviati al processo stesso. Alcune trap sono causate da comportamenti errati del processo stesso, e immediatamente inviate al processo che normalmente reagisce terminando.

> *per esempio tentativi di divisione per zero (SIGFPE), indirizzamento errato degli array (SIGSEGV), tentativo di eseguire istruzioni privilegiate (SIGILL), etc..*

Unix Interrupt

Gli *interrupt* sono segnali inviati ad un processo da un agente esterno: l'utente o un altro processo:

> *Utente*

- `CTRL + C`: invia `SIGINT`
- `CTRL + Z`: invia `SIGSTOP`
- comando `kill`: `kill -s SIGNAL PID`

  > *Altro processo:*

- System call kill: `kill(PID, SIGNAL)`

Impostare i signal handlers

Invece di accettare l'azione di default per un particolare segnale, *un programma può modificare l'azione da intraprendere al momento della consegna (delivery) del segnale*. Questo è noto come impostazione dell'handler del segnale. Un programma può impostare una dei seguenti handler del segnale:

- L'azione di default dovrebbe essere intrapresa. Utile per cancellare precedenti modifiche della disposizione del segnale che modificavano la disposizione di default.
- Il segnale è ignorato. Utile per un segnale la cui disposizione sarebbe quella di terminare il processo.

- Viene eseguito un signal handler.

Signal handlers

Un signal handler (o gestore di segnali) è una funzione, scritta dal programmatore, che esegue azioni appropriate in risposta alla ricezione di un segnale. Per esempio, la shell ha un gestore per il segnale `SIGINT` (generato dal carattere interrupt, Control-C) che causa il suo blocco (stop) e la restituzione del controllo al ciclo di input principale: in questo caso all'utente viene presentato il prompt della shell.

La notifica al kernel del fatto che deve essere invocata una funzione handler è detto installare o stabilire un signal handler.

Quando un handler è invocato in risposta alla ricezione di un segnale, diciamo che il segnale è stato gestito (handled) o intercettato (caught).

SIGABRT. Un processo riceve questo segnale quando invoca la funzione `abort()`. Di default questo segnale termina il processo con un core dump. Questo produce l'effetto della chiamata `abort()`, che produce un core dump a fini di debug.

> *6 SIGABRT create core image abort program (formerly SIGIOT)*

SIGALRM. Il kernel genera questo segnale al momento del raggiungimento dello zero di un timer impostato da una chiamata ad `alarm()` o `setitimer()`.

> *14 SIGALRM terminate process real-time timer expired*

SIGCHLD. Segnale inviato dal kernel a un processo genitore quando uno dei figli termina (chiamando `exit()`, o ucciso da un qualche segnale), o viene bloccato o viene risvegliato da un segnale.

> *20 SIGCHLD discard signal child status has changed*

SIGCONT. Quando viene inviato a un processo bloccato (*stopped*), questo segnale causa il risveglio del processo (*resume*), cioè che il processo venga schedulato per successivamente essere eseguito. Quando è ricevuto da un processo che non è bloccato, questo segnale è ignorato di default. Un processo può intercettare

questo segnale, in modo da eseguire qualche azione particolare al momento della ripresa dell'esecuzione.

> *19 SIGCONT discard signal continue after stop*

SIGINT. Quanto l'utente digita il carattere di interrupt (`<Ctrl> + <C>`), il terminale invia questo segnale al gruppo del processo in foreground. L'azione di default per questo segnale è terminare il processo.

> *2 SIGINT terminate process interrupt program*

SIGKILL. È il segnale sicuro di kill. *Non può essere bloccato, ignorato, o intercettato da un handler* di conseguenza, *termina sempre un processo*.

> *9 SIGKILL terminate process kill program*

SIGPIPE. Segnale generato quando un processo tenta di scrivere su un pipe o un FIFO per il quale non c'è un corrispondente processo lettore. Questo normalmente occorre perché il processo lettore ha chiuso il proprio file descriptor per il canale IPC.

> *13 SIGPIPE terminate process write on a pipe with no reader*

SIGSEGV. Segnale generato quando un programma tenta un riferimento in memoria non valido. Il riferimento può non essere valido perché la pagina riferita non esiste (per esempio, giace in un'area non mappata, fra heap e stack), oppure il processo ha tentato di modificare una locazione in read-only memory (il segmento di testo del programma o una regione di memoria marcati come disponibili in sola lettura), o il processo ha tentato di accedere a una parte della memoria del kernel durante l'esecuzione in user mode.
In C, questi eventi spesso derivano dalla dereferenziazione di un puntatore che contiene un 'bad address' (come un puntatore non inizializzato) o dal passaggio di un argomento non valido in una chiamata a funzione.
Il nome del segnale deriva da segmentation violation.

> *11 SIGSEGV create core image segmentation violation*

SIGSTOP. Segnale per il blocco (stop) sicuro. *Non può essere bloccato, ignorato, o intercettato da un handler*; quindi questo segnale *blocca sempre un processo*.

> *17 SIGSTOP stop process stop (cannot be caught or ignored)*

SIGTERM. Segnale standard utilizzato per terminare un processo; segnale inviato di default dai comandi `kill` e `killall`. Gli utenti a volte inviano esplicitamente il segnale `SIGKILL` a un processo, usando `kill –KILL` o `kill –9`.
In generale, questo è un errore. Un'applicazione ben progettata deve avere un handler per `SIGTERM` che consenta una '*graceful*' exit, che consenta di pulire i file temporanei e di rilasciare le altre risorse. L'uccisione di un processo con `SIGKILL` bypassa l'handler di `SIGTERM`, e quindi si dovrebbe sempre prima cercare di terminare un processo con `SIGTERM`, e tenere `SIGKILL` come ultima scelta per terminare i processi che non rispondono a `SIGTERM`.

> *15 SIGTERM terminate process software termination signal*

SIGTRAP. Segnale utilizzato per implementare i breakpoint in fase di debugging e per la tracciatura delle system call. Cercare `ptrace()` sul manuale per ulteriori informazioni.

> *5 SIGTRAP create core image trace trap*

SIGTSTP. Segnale per lo stop, inviato per bloccare il gruppo di processi in foreground quando l'utente digita il carattere di sospensione (`<Ctrl> + <Z>`) sulla tastiera.
il nome di questo segnale deriva da "terminal stop".

> *18 SIGTSTP stop process stop signal generated from*

SIGUSR1 e SIGUSR2 sono disponibili per fini specificati dal programmatore. Il kernel non genera mai questi segnali per un processo.
I processi possono utilizzare questi segnali per notificarsi a vicenda eventi, o per sincronizzarsi.

> *30 SIGUSR1 terminate process User defined signal 1*
> *31 SIGUSR2 terminate process User defined signal 2*

Invio di segnali

kill(), raise(), alarm()

Tramite linea di comando

Si usa il comando `kill`:

- `kill -INT <PID>`
- `kill -SIGINT <PID>`
- `kill -2 <PID>`
  se il segnale non è specificato, allora viene inviato `SIGTERM`.

System Call kill()

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

L'argomento pid identifica uno o più processi a cui inviare il segnale; pid può essere interpretato in 4 modi:

- `pid > 0`: il segnale è inviato al processo identificato da pid.
- `pid == 0`: il segnale è inviato a ogni processo nello stesso gruppo del chiamante, chiamante incluso.
- `pid < -1`: il segnale è inviato a tutti i processi nel gruppo del processo il cui ID è uguale al valore assoluto di pid. Inviare un segnale a tutti i processi nel gruppo di un processo è utile nel controllo dei job effettuato con la shell.
- `pid == -1`: (*broadcast signal*) il segnale è inviato a tutti i processi per i quali il processo ha i permessi di inviare un segnale, eccetto `init` (che ha pid 1) ed il chiamante.
  Se l'utente non è super user, il segnale è inviato a tutti i processi con stesso `uid` dell'utente, escluso il processo che invia il segnale.
- Se nessun processo corrisponde al pid predefinito, `kill()` fallisce e setta `errno` a `ESRCH` (“No such process”).
- Se il processo esiste, ma non si hanno i permessi per inviare un segnale, allora `errno = EPERM`.

Verifica dell'esistenza di un processo

Se l'argomento `sig` è settato a 0 (detto null signal), non è inviato alcun segnale. In questo caso `kill()` esegue unicamente un controllo degli errori per vedere se è possibile inviare segnali al processo: il null signal può essere utilizzato per testare se un processo con un certo pid esiste. Se la chiamata va a buon fine, sappiamo che il processo esiste.

System call raise()

Permette di inviare un segnale al processo che invoca la system call.

```
#include <signal.h>
int raise(int sig);
// Returns 0 on success, -1 on error
```

System call alarm()

Permette di inviare un segnale `SIGALRM` al processo stesso che invoca la system call dopo un numero di secondi:

```
#include <signal.h>
usigned int alarm (unsigned int sec);
```

Ritorna:

- 0 se non ci sono precedenti invocazioni di alarm pendenti
- il numero di secondi che mancano alla scadenza del prossimo allarme, se ci sono invocazioni di alarm pendenti

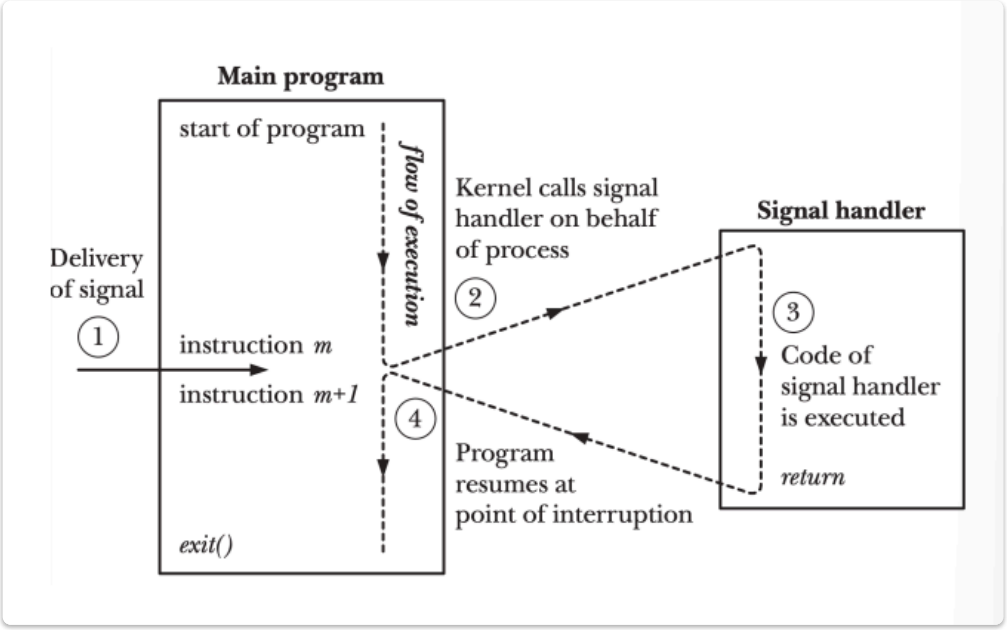
Cambiare l'handler dei segnali

Nei sistemi UNIX vi sono due modi per cambiare l'handler di un segnale: `signal()` e `sigaction()`. La system call `signal()` è l'API originale per assegnare l'handler di un signal, e fornisce un'interfaccia più semplice di `sigaction()`.

> ⚠ **Vi sono differenze nel comportamento di `signal()` fra le varie implementazioni UNIX.**

Signal Handlers
Un signal handler è una funzione chiamata quando un processo riceve uno specifico segnale.
L'invocazione di un handler può interrompere il flusso principale del programma in qualsiasi momento.

1. Il kernel chiama l'handler da parte del processo
2. Quando l'handler restituisce, l'esecuzione del programma riprende dal punto in cui l'handler lo aveva interrotto.



sigaction()

> *man 2 sigaction*

Gestione dei segnali più completa e robusta rispetto all'uso di `signal()` (che tra l'altro non va usata).

```
#define _GNU_SOURCE
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

Parametri:

- `signum`: segnale che deve essere gestito
- `act`: nuovo handler per il segnale (se NULL, nessun cambiamento)
- `oldact`: puntatore al vecchio handler

```
struct sigaction new;
struct sigaction old;

// Set new handler to new
sigaction(signum, new, NULL);

// current handler in old
sigaction(signum, NULL, old);
```

```
// do both
sigaction(signum, new, old);
```

Sigaction structure

Principali elementi:

- `sa_handler` : puntatore alla funzione di handling
- `sa_mask` : maschera per indicare i segnali bloccati *durante* l'esecuzione dell'handler
- `sa_flags` : maschera bitwise (*or-style*) che modifica il comportamento del gestore del segnale.

```
#define _GNU_SOURCE
#include <signal.h>

struct sigaction {
    void (*sa_handler) (int signum);
    sigset_t sa_mask;
    int sa_flags;
    //Plus others for advanced use cases
}
```

gestire più segnali con un unico handler

Il costrutto switch può essere utilizzato per discriminare il tipo di segnale che l'handler ottiene come argomento.

```
void sa_handler (int signum){
    switch (signum) {
        case SIGINT:
            /* handle SIGINT*/
            break;
        case SIGALRM:
            /* handle SIGALRM*/
            breeak;
        // And so on...
    }
}
```

- Le funzioni di gestione handler vengono ereditate dai processi figli dopo una fork
- Le variabili globali definite nel programma sono visibili sia dalle funzioni handler che dal resto del programma
    - Questo può essere utile per modificare il valore di una variabile globale simulando un cambio di stato del programma che verrà poi utilizzato durante l'esecuzione *"normale"* del codice, ma...

handler definiti dall'utente e variabili globali

Le variabili globali possono essere utili, ma cosa succede quando un handler modifica il valore di una variabile globale il cui valore non dovrebbe essere modificato?

- Alcune system call utilizzano strutture dati globali e quindi il loro utilizzo all'interno di un handler può generare problemi

> ⚠ **Ad esempio** `printf()`
> Se un segnale interrompe l'esecuzione di una `printf()`, che poi viene anche usata all'interno dell'handler, può verificarsi una corruzione del buffer o un output misto.
>
> per maggiori informazioni: `man signal-safety`

Fortunatamente vi sono funzioni che sono definite in `libc` *"AS-Safe"* (Asynchronous Signal-Safe). É dunque bene controllare sempre dalla documentazione se ogni system call utilizzata in un handler é *AS-Safe*

> **Ad esempio** `write()`.

`errno` é una variabile globale ed il suo valore potrebbe essere sovrascritto durante l'esecuzione di un handler, é quindi buona pratica salvare e ripristinare il valore di `errno` nell'handler.

Signal mask

Quando un segnale `signum` é consegnato ad un processo, durante l'esecuzione del suo handler, il segnale `signum` é bloccato, a meno che il flag `SA_NODEFER` sia settato nella `sigaction`.

```
struct sigaction sa;
sigset_t my_mask;

sa.sa_handler = handle_signal;
sa.sa_flags = SA_NODEFER; // allow nested invocations
sigemptyset(&my_mask); // do not mask any signal
sa.sa_mask = my_mask;

sigaction(SIGUSR1, &sa, NULL); // set the handler
```

La maschera é la collezione di segnali attualmente blocked.

- ogni processo ha la propria maschera di segnali: un nuovo processo eredita la maschera del genitore
  Le maschere sono del tipo `sigset_t`
- le funzioni per la manipolazione dei set sono:

  🔥 `man sigsetops`

```
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signum);
int sigdelset (sigset_t *set, int signum);
int sigismember (const sigset_t *set, int signum);
```

Impostare la maschera dei segnali di un processo

```
#include <signal.h>
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
// returns 0 if successful, otherwise returns -1 and sets errno
```

Per impostare la maschera di segnali bloccati durante l'esecuzione dei processi si una la system call `sigprocmask()`

- l'argomento `how` può assumere i seguenti valori:
  - `SIG_BLOCK`: i segnali nel set sono aggiunti a quelli bloccati
  - `SIG_UNBLOCK`: i segnali nel set sono rimossi dalla maschera esistente

    - `SIG_SETMASK`: il set diventa la nuova maschera del segnale
- `oldset` é la vecchia maschera

```
static int segnale_ricevuto = 0;

int main (int argc, char *argv[]) {
    sigset_t orig_mask;
    sigset_t mask;
    struct sigaction act;

    printf("process pid: %d\n", getpid());
    memset (&act, 0, sizeof(act));
    act.sa_handler = mio_handler;

    if (sigaction(SIGTERM, &act, 0))
        // gestione errore
    sigemptyset (&mask);
    sigaddset (&mask, SIGTERM);

    if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0)
        // gestione errore
    sleep (20);
    if (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0)
        // gestione errore
    sleep (1);
    if (segnale_ricevuto) puts (“signal ricevuto!!!”);

    return 0;
}
```

L'handler può comunque essere interrotto da altri segnali (o dallo stesso segnale, nel caso `SA_NODEFER` sia settato).
Quando l'handler termina, l'insieme di segnali bloccati é reimpostato al suo valore precedente la sua esecuzione, indipendentemente dalle possibili manipolazioni dei segnali bloccati eventualmente presenti nell'handler.

```
struct sigaction sa;
sigset_t my_mask;

sa.sa_handler = mio_handler; // handler
sa.sa_flags = 0; // No special behaviour
```

```
// Create an empty mask
sigemptyset(&my_mask); // Do not mask any signal
sigaddset(&my_mask, signal_to_mask_in_handler); // for ex:
SIGTERM
sa.sa_mask = my_mask;

sigaction(SIGINT, &sa, NULL); // Set the handler
```

Merged signals

Se un segnale viene generato mentre un altro segnale (stesso segnale) é in stato pending, il nuovo segnale e quello pending sono accorpati in uno.

La presenza di un segnale pending é gestita solo con un flag, non da un numero di segnali in stato pending.

- un gestore di segnali non può essere utilizzato per contare il numero di segnali ricevuti

Delivery di segnali a processi sospesi

pause(), sleep() e nanosleep()

```
#include <unistd.h>
int pause (void);
```

```
#include <unistd.h>
unsigned int sleep (unsigned int seceonds);
```

```
#include <time.h>
struct timespec my_time;
my_time.tv_sec = 1;
my_time.tv_nsec = 234567000;

nanosleep (&my_time, NULL);
```

```
man 7 signal
```

All'arrivo (asincrono) di un segnale

1. Lo stato del processo è salvato (registers, etc.)
2. La funzione di handler è eseguita
3. Lo stato iniziale del processo è ripristinato

Quando un processo è in attesa su wait() (o altre system call), o sospeso con pause() o sleep():

1. il processo non è in esecuzione (è sospeso su qualche system call)
2. la funzione dell'handler è eseguita normalmente
3. quando l'handler ritorna:
   - A. la syscall restituisce un errore, con `errno` settato a `EINTR`;
   - B. la syscall viene automaticamente ripresa.

A o B? dipende dal sistema operativo, e dal flag `SA_RESTART` nella syscall sigaction().

- Il comportamento di default è A (aborting)
- Se il flag `SA_RESTART` è settato in `sa_flags`, si ha invece un restart della system call che aveva generato l'attesa
- Il comportamento dipende dalla system call, purtroppo
- Nella sezione «Interruption of system calls and library functions by signal handlers» di «man 7 signal» ci sono le indicazioni puntuali

Gestione asincrona dei segnali

> ⚠ **NON UTILIZZATA DURANTE IL CORSO**

La gestione asincrona dei segnali é possibile

- un processo può attendere la ricezione di un particolare segnale

Si fa uso delle system call `sigwaitinfo()`, `sigtimedwait()`, `sigwait()`

PIPE E FIFO

I vari strumenti che UNIX offre per la comunicazione e la sincronizzazione possono essere suddivisi in tre ampie categorie funzionali:

- *Comunicazione*: facilities utilizzate per lo scambio di dati fra processi.
- *Sincronizzazione*: facilities utilizzate per sincronizzare le azioni dei processi.

- *Segnali*: sebbene i segnali siano nati prevalentemente con altri fini, in alcune circostanze possono essere utilizzati come strumenti di sincronizzazione.

Communication facilities

Data-transfer facilities: l'elemento fondamentale che distingue questi strumenti è la nozione di lettura e scrittura

- Per comunicare, un processo scrive i dati alla facility per l'IPC e un altro processo legge questi dati
- Questi strumenti richiedono due trasferimenti dati fra la memoria utente e quella del kernel: un trasferimento durante la scrittura ed un trasferimento durante la lettura

Memoria condivisa: la memoria condivisa permette ai processi di scambiarsi le informazioni mettendole in una regione della memoria condivisa fra i processi.

- un processo può rendere i dati disponibili per gli altri processi collocandoli in una regione di memoria condivisa
- poiché la comunicazione non richiede system call o trasferimento di dati fra la memoria utente e quella del kernel, la memoria condivisa è uno strumento di comunicazione molto veloce.

Data-transfer facilities

Le data transfer facilities possono essere ulteriormente suddivise nelle seguenti sottocategorie:

- *Byte stream*: i dati scambiati per mezzo di pipe, FIFOs, e datagram sockets sono uno stream di byte.
    - ogni operazione di lettura può leggere un numero arbitrario di byte, senza considerare la dimensione dei blocchi scritti dallo scrivente.
    - questo modello riflette il tradizionale modello di UNIX in cui il file è visto come una sequenza di byte.
- *Messaggio*: i dati scambiati con le code di messaggi, e i socket hanno la forma di messaggi delimitati.
    - ogni operazione di lettura legge un intero messaggio, così come scritto dal processo scrivente.
    - non è possibile leggere parzialmente un messaggio, lasciando il resto sulla IPC facility, e non è possibile leggere molteplici messaggi con una

singola operazione di lettura.

Le Data-transfer facilities sono distinte dalla memoria condivisa per alcune caratteristiche generali:

- sebbene le data-transfer facilities possano avere molteplici lettori, le operazioni di lettura sono distruttive. Una operazione read consuma i dati, e i dati non sono più disponibili per altri processi.
- La sincronizzazione fra processo lettore e scrittore è automatica. Se un lettore tenta di consumare dati da una facility che attualmente non ne contiene, (di default) l'operazione di lettura si bloccherà finché un processo non avrà scritto dei dati su quella facility.

Memoria condivisa

Sebbene la memoria condivisa fornisca una comunicazione veloce, questo vantaggio è bilanciato dalla necessità di sincronizzare le operazioni sulla memoria condivisa.

- per esempio, un processo non dovrebbe cercare di accedere a una struttura dati presente nella memoria condivisa mentre un altro processo la sta modificando.

il semaforo è lo strumento di sincronizzazione abitualmente utilizzato con la memoria condivisa.

- i dati presenti nella memoria condivisa sono visibili a tutti i processi che condividono quel segmento di memoria, diversamente dalla semantica distruttiva delle operazioni di lettura messe a disposizione dalle data-transfer facilities.

semafori

Un *semaforo* è un intero mantenuto dal kernel, il cui valore non può divenire minore di 0.

Un processo può decrementare o incrementare il valore di un semaforo. Se viene fatto un tentativo di decrementarne il valore sotto lo 0, il kernel blocca l'operazione finché il valore del semaforo aumenta ad un livello che permetta di eseguire l'operazione.
In alternativa, il processo può richiedere nonblocking operation, in questo caso,

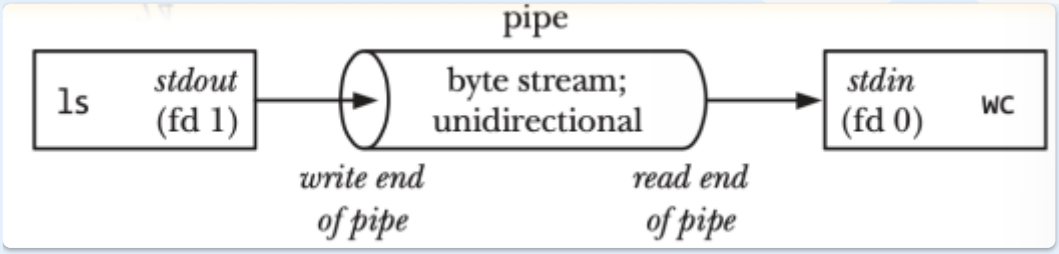
invece di bloccarlo, il kernel provoca una restituzione immediata con un errore che indica che non è stato possibile eseguire l'operazione immediatamente.

PIPE

Flusso di byte unidirezionale, forniscono una soluzione ad un problema frequente: avendo creato due processi per eseguire programmi diversi (comandi), come può la shell fare in modo che l'output prodotto da un processo sia utilizzato come input per l'altro processo?
Le FIFO sono una variazione del concetto di pipe. La differenza sostanziale è che le FIFO possono essere utilizzate per la comunicazione fra processi qualsiasi.

esempio su shell: `ls -al | wc -l`

Per eseguire questi comandi, la shell crea due processi, che eseguono ls e wc, rispettivamente. Questo è fatto usando la `fork()` e la `exec()`.



i due processi sono collegati alla pipe: il processo scrittore `ls` ha il proprio standard out (file descriptor 1) collegato con il write end del pipe, mentre il processo lettore `wc` ha il proprio standard input (file descriptor 0) collegato al read end del pipe.

NB: i due processi sono completamente ignari dell'esistenza del pipe: semplicemente leggono e scrivono da/su descrittori di file standard.

Una pipe è un flusso di byte: usando una pipe, non facciamo riferimento ad alcun concetto di messaggio o di delimitazione di messaggio. Il processo che legge da una pipe può leggere blocchi di qualsiasi dimensione, indipendentemente dalla dimensione dei blocchi scritti dal processo che scrive. I dati passano attraverso la pipe in sequenza: i byte sono letti nello stesso ordine in cui sono stati scritti. Non è possibile accedere ai dati in maniera casuale.
(ad es: utilizzare `lseek()` provoca un errore con `errno` = `EPIPE`).

I tentativi di leggere da una pipe vuota restano bloccati finché almeno un byte non è stato scritto sulla pipe. Se il write end di un pipe viene chiuso, un processo che legge dalla pipe riceverà il codice end-of-file (i.e. `read()` restituirà 0) una volta che avrà letto tutti i dati presenti nella pipe.

Le pipe sono unidirezionali. I dati possono viaggiare solo in una direzione. Un'estremità (end) del pipe è utilizzato in scrittura e l'altro in lettura.

capacità limitata

Una pipe è semplicemente un buffer mantenuto in memoria.
Questo buffer ha una capacità massima (`PIPE_BUF`). Una volta che una pipe è piena, ulteriori tentativi di scrittura si bloccano finché il lettore rimuove alcuni dati dalla pipe.
In generale, un applicazione non dovrebbe avere bisogno di conoscere la capacità della pipe.

Se vogliamo evitare ai processi scrittori di restare bloccati, è necessario che i processi che leggono dalla pipe siano progettati in modo da leggere i dati appena questi sono disponibili. In teoria, non ci sono motivi per cui una pipe non debba utilizzare capacità minime, fino al buffer costituito da un solo byte.
La ragione per usare buffer di dimensioni maggiori è l'efficienza: ogni volta che uno scrittore riempie la pipe, il kernel deve eseguire un context switch per consentire al lettore di essere schedulato in modo che possa prelevare qualche dato dalla pipe. L'utilizzo di un buffer di dimensione maggiore comporta la riduzione del numero di context switch.

```c
#include <unistd.h>
int pipe (int filedes[2]);
// returns 0 on success, -1 on error
```

la system call `pipe()` crea una nuova pipe, se va a buon fine, la chiamata alloca un array (`filedes`) contenente due descrittori di file aperti.
Un estremo è aperto in lettura (`filedes[0]`) e uno in scrittura (`filedes[1]`)
Come con qualsiasi descrittore di file, possiamo utilizzare le system call `read()` e `write()` per eseguire le operazioni di I/O sulla pipe.

Di norma si utilizzano le pipe per permettere la comunicazione fra due processi.
Per collegare due processi con una pipe, eseguiamo prima una `pipe()` e poi una

`fork()`. Con la `fork()`, il processo figlio eredita copia dei descrittori di file dei genitori.

Subito dopo la `fork()`, un processo chiude il proprio descrittore per l'estremità in scrittura, e l'altro chiude il proprio descrittore per l'estremità in lettura. Per esempio, se il genitore deve inviare dei dati al figlio, deve chiudere l'estremità aperta in lettura, `filedes[0]`, mentre il figlio deve chiudere `filedes[1]`.

Lettura da una pipe

Un operazione di lettura consuma i dati presenti nel buffer della pipe.

- Se i dati sono presenti, la read ritorna il numero di byte letti, che saranno memorizzati nella variabile buffer.
- Se i dati non sono presenti, ma almeno un file descriptor in scrittura è ancora aperto, allora la read è bloccante. Se nessun file descriptor in scrittura è aperto, allora la read ritorna il valore 0.

```
char buffer[100];
int n_bytes;

n_bytes = read(filedes[0], buffer, sizeof(buffer)):
```

Scrittura su una pipe

Dopo un operazione di scrittura su una pipe:

- se il buffer può contenere i dati passati come parametro, la `write` ritorna il numero di byte scritti
- se il buffer della pipe è pieno, la `write` è bloccante (in attesa di una `read`)
- se nessun file descriptor in lettura è ancora aperto, un segnale `SIGPIPE` è generato (per notificare che i dati appena scritti non saranno mai leggibili).

```
char buffer[100];
int n_bytes;
n_bytes = write(filedes[1], buffer, sizeof(buffer));
```

Chiusura dei descrittori inutilizzati (lettore)

I file descriptors inutilizzati in lettura e in scrittura devono essere chiusi.
Il processo che legge dalla pipe chiude il proprio write descriptor, così che quando l'altro processo completa il proprio output e chiude il proprio descrittore write, il lettore riceve, dopo aver consumato tutti i byte nel buffer, il valore di ritorno 0 (che indica un `EOF`)

- Se invece il processo lettore non chiude la propria estremità aperta in scrittura, dopo che l'altro processo avrà chiuso il proprio descrittore write, il lettore non riceverà l'end-of-file neppure dopo avere letto tutti i dati dalla pipe.
- In questo caso, una `read()` si bloccherebbe in attesa di dati (che sappiamo non arriveranno!) perché il kernel sa che c'è ancora almeno un descrittore di file aperto per la pipe

Chiusura dei descrittori inutilizzati (scrittore)

Il processo scrittore chiude l'estremità aperta in lettura della pipe per una ragione diversa. Quando un processo tenta di scrivere su una pipe per il quale nessun processo ha un descrittore aperto in lettura, il kernel invia il segnale SIGPIPE al processo scrittore.
Un processo può organizzarsi per intercettare o ignorare tale segnale; in questo caso la `write()` sulla pipe fallisce con un errore `EPIPE` (broken pipe).

- Ricevere il segnale `SIGPIPE` o ricevere l'errore `EPIPE` è un'indicazione utile in merito allo status della pipe, ed è la ragione per cui i descrittori aperti in lettura dovrebbero essere chiusi.

Considerazioni

Le operazioni di lettura e scrittura sono atomiche (importante nel caso in cui si avessero molteplici scrittori e lettori).

La dimensione del buffer della pipe è indicata in `PIPE_BUF` di `<limits.h>`

Se un processo è bloccato in attesa di una lettura o scrittura e viene interrotto da un segnale, l'opzione ritorna -1 con `errno = EINTR`.

Comunicazione tra processi NON padre-figlio

Le pipe possono essere usate per la comunicazione fra (due o più) processi parenti, supponendo che la pipe sia creata da un antenato comune ad entrambi

e prima della serie di chiamate `fork()` con cui sono stati creati i vari figli.
Per esempio, una pipe potrebbe essere usata per mettere in comunicazione un processo ed un processo nipote (grandchild). Il primo processo crea la pipe, e quindi effettua una `fork()` e il figlio effettua un ulteriore `fork()` creando così il nipote.
Un altro scenario frequente è l'utilizzo di una pipe per la comunicazione fra due processi fratelli (siblings): il loro genitore crea la pipe, e quindi crea i due figli.

copiare un file descriptor su un altro

```c
#include <unistd.h>

int dup2 (int fd_src, int fd_dst);
```

`dup2()` copia il file descriptor `fd_src` sul file descriptor `fd_dst`.
Se `fd_dst` è stato precedentemente aperto, allora `dup2()` provvede a chiuderlo.

Esempio: se `fd[1]` e il write end di scrittura di una pipe, `dup2(fd[1], 1)` redirige tutto quello che prima veniva diretto sullo `stdout` verso la pipe.
Una `printf` avrà quindi l'effetto di scrivere sulla pipe e non su `stdio`.

FIFO

Una FIFO è simile ad una pipe, con la differenza che una FIFO ha un nome all'interno del file system ed è aperta nello stesso modo di un file.
Questo permette ad una FIFO di essere utilizzata per comunicazioni fra processi non imparentati (es: un client ed un server)

Una volta che la FIFO è stata aperta possiamo utilizzare le stesse system call dell'I/O utilizzata con le pipe e gli altri file (`read()`, `wirte()`, `close()`).

Come per le pipe, anche le FIFO hanno un estremità aperta in scrittura e una aperta in lettura, e come per le pipe, i dati sono letti nello stesso ordine con cui sono stati scritti. Questa caratteristica conferisce alle FIFO il loro nome FirstInFirstOut. Le FIFO sono anche note come *named pipes*.

Creare una fifo (command line)

Possiamo creare una FIFO dalla shell con il comando:
`fifo [ –m mode ] pathname`

Il `pathname` e il nome della FIFO che si intende creare, e l'opzione -m specifica i permessi analogamente al comando `chmod`.
Listato con `ls -l`, una FIFO è mostrata con il carattere `p` nella prima colonna.

Creare una fifo (in C)

La funzione `mkfifo()` crea una nuova FIFO con il `pathname` dato.
L'argomento mode specifica i permessi per la nuova FIFO. Tali permessi sono specificati mettendo in OR varie possibile costanti.

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

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

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

L'utilizzo di una FIFO serve ad avere un processo lettore e uno scrittore alle due estremità.
Di default, l'apertura di una FIFO in lettura (flag `O_RDONLY` della `open()`) si blocca finché un altro processo apre la FIFO in scrittura (flag `O_WRONLY` della `open()`). Per contro, l'apertura della FIFO in scrittura si blocca finché un altro processo non apre la FIFO in lettura.
In altri termini, l'apertura di una FIFO sincronizza i processi lettori e scrittori.
Se l'altra estremità della FIFO è già aperta (per esempio nel caso in cui una coppia di processi hanno già aperto ciascuna estremità della FIFO), la `open()` va immediatamente a buon fine.

Code di messaggi (System V IPC)

Le code di messaggi possono essere utilizzate per scambiare messaggi fra processi. Le code di messaggi sono simili alle pipe, da cui differiscono per 2 aspetti.

- i confini dei messaggi sono delimitati, così che i lettori e gli scrittori comunicano in unità di messaggi, e non in stream di byte privi di delimitazioni interne.
- ciascun messaggio contiene un membro type di tipo intero, ed è possibile selezionare i messaggi per tipo, piuttosto che leggerli nell'ordine in cui sono stati scritti.

Semafori (System V IPC)

I semafori permettono a molteplici processi di sincronizzare le proprie azioni. Un semaforo è un valore intero mantenuto dal kernel visibile a tutti i processi che hanno i permessi necessari. Un processo indica ai propri pari che sta eseguendo una qualche azione facendo una modifica al valore del semaforo.

Memoria Condivisa (System V IPC)

La memoria condivisa consente a molteplici processi di condividere lo stesso segmento di memoria. Poiché l'accesso allo spazio della memoria utente è un'operazione veloce, la memoria condivisa è uno dei più veloci strumenti per l'IPC: una volta che un processo ha aggiornato la memoria condivisa, la modifica è immediatamente visibile agli altri processi che condividono lo stesso segmento.

Gestione della memoria da linea di comando

(Vedi `man` per istruzioni dettagliate)

`ipcs` : elenca le risorse IPC definite
`ipcs -l` : elenco dei limiti definiti per le risorse IPC
`ipcmk` : crea una nuova risorsa IPC
`ipcrm` : elimina una risorsa IPC

Specifichiamo *key* (oppure *id*).
q indica le code di messaggi

s indica i semafori
m indica memoria condivisa

creazione / apertura

Ciascun meccanismo delle System V IPC ha associato una system call get (`msgget()`, `semget()` o `shmget()`) che è l'analogo della system call `open()` utilizzata per i file.

Data una *key* intera (analoga ad un filename) la chiamata get:

- crea un nuovo oggetto IPC con la key indicata e restituisce un identificatore unico per quell'oggetto, oppure
- restituisce l'identificatore di un oggetto IPC già esistente a avente la stessa *key*.

Esempio di creazione di una coda di messaggi con `msgget()`

```
id = msgget (key, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1) errExit("msgget");
```

Come per tutte le altre system call get, *key* è il primo argomento, e l'identificatore è restituito come risultato della funzione.
Specifichiamo i permessi di accesso al nuovo oggetto come ultimo argomento (flags), utilizzando le seguenti costanti in OR:

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

Ogni processo che desideri accedere allo stesso oggetto IPC esegue una chiamata get, specificando la stessa key per ottenere lo stesso identificatore per quell'oggetto.

Se non esiste un oggetto IPC corrispondente alla key, ed è stata specificata la costante `IPC_CREAT` come parte dei flag, la get crea un nuovo oggetto IPC.

Un processo può garantire di essere il creatore di un oggetto IPC specificando il flag `IPC_EXCL`: se viene specificato, e l'oggetto IPC corrispondente a key esiste già, la get fallisce con l'errore `EEXIST`.

Cancellazione di oggetti IPC

La system call `ctl` (`msgctl()`, `semctl()`, `shmctl()`) per ciascun meccanismo di IPC esegue un gran numero di operazioni di controllo sull'oggetto.
Mentre molte di queste operazioni sono specifiche dei vari meccanismi di IPC, alcune sono comuni a tutti.

- per esempio, una generica operazione di controllo è `IPC_RMID`, utilizzata per cancellare un oggetto.

Per le code di messaggi e i semafori, la cancellazione degli oggetti IPC è immediata, e qualsiasi informazione contenuta all'interno dell'oggetto è distrutta, a prescindere dal fatto che qualche altro processo stia ancora utilizzando quell'oggetto.

Per gli oggetti legati alla memoria condivisa, vi è un differente comportamento:

- in seguito alla chiamata `shmctl(id, IPC_RMID, NULL)`, il segmento di memoria condivisa è rimosso solo dopo che tutti i processi che lo utilizzano effettuano il detach (molto più simile alla situazione della cancellazione di un file).

Persistenza degli oggetti IPC

Gli oggetti IPC hanno una kernel persistence: dopo essere stati creati, continuano ad esistere finché sono esplicitamente cancellati o il sistema viene spento.
Tale proprietà degli oggetti IPC fornisce alcuni vantaggi:

- è possibile per un processo creare un oggetto, modificarne lo stato ed uscire, lasciando che l'oggetto resti accessibile da altri processi iniziati successivamente.
  ma anche svantaggi:
- esistono limiti di sistema sul numero massimo di IPC di ogni tipo

Il vero problema degli oggetti IPC è che sono connectionless: il kernel non tiene traccia di quali processi hanno un oggetto aperto.

- Quando si cancella una coda di messaggi, un applicazione con molti processi può non essere agevolmente in grado di determinare quale sarà l'ultimo processo a richiedere l'accesso all'oggetto e quindi quando l'oggetto può essere cancellato senza problemi.

IPC keys

- Le keys dei meccanismi IPC di System V sono valori interi rappresentati con il tipo `key_t`.
- La chiamata `...get` mappa una key sul corrispondente identificatore IPC intero.
- Queste chiamate garantiscono che:
  - se creiamo un nuovo oggetto, quell'oggetto ha un identificatore unico
  - se specifichiamo la key di un oggetto esistente, otteniamo sempre lo stesso identificatore per quell'oggetto

Come si genera la key unica, tale da garantirci di non ottenere accidentalmente l'identificatore di un oggetto IPC esistente, utilizzato da qualche altra applicazione?

1. scelta casuale di un valore intero, che è tipicamente memorizzato in un header file incluso da tutti i programmi che usano l'oggetto IPC.
2. Utilizzo della costante `IPC_PRIVATE` come valore della key nell'invocazione alla get al momento della creazione dell'oggetto , che produce sempre un oggetto con una chiave unica.
3. Utilizzo della funzione `ftok()` per generare una key molto probabilmente unica.

Generazione di una key con `IPC_PRIVATE`

```
id = msgget (IPC_PRIVATE, S_IRUSR | S_IWUSR);
```

In questo caso non è necessario specificare i flag `IPC_CREAT` o `IPC_EXCL`.
Questa tecnica è particolarmente utile in applicazioni con molti processi in cui il processo padre crea l'oggetto IPC prima di eseguire la `fork()`, con il risultato che il figlio eredita l'identificatore dell'oggetto IPC.

È possibile utilizzare questa tecnica anche in applicazioni client-server (che coinvolgono processi non collegati), ma i client devono avere un mezzo per ottenere gli identificatori degli oggetti IPC creati dal server (e viceversa).

- Per esempio, dopo aver creato un oggetto IPC, il server potrebbe scrivere il proprio identificatore su un file, che potrebbe essere letto dal client.

Generazione di una key con `ftok()`

```
#include <sys/ipc.h>
key_t ftok (char *pathname, int proj);
// returns integer key onm success, or -1 on error
```

Questo valore di key è generato dal pathname fornito e dal valore di `proj` utilizzando un algoritmo definito a livello di implementazione.
Il fine del valore `proj` è di consentire di generare diverse key a partire dallo stesso file, è utile quando un'applicazione deve creare vari oggetto IPC dello stesso tipo.

> Storicamente, l'argomento `proj` era di tipo char, ed è ancora spesso specificato come tale nelle chiamate a `ftok()`.

```
key_t key;
int id;
key = ftok("./mydir/myfile", 'x');
//...
id = msgget (key, IPC_CREAT | S_IRUSR | S_IWUSR);
//...
```

Gestione dei permessi su un oggetto IPC

Il kernel mantiene una struttura dati per ogni istanza di un oggetto IPC.
La forma di questa struttura dati varia a seconda del meccanismo (code di

messaggi, semafori o memoria condivisa) ed è definito nell'header file corrispondente a ciascun meccanismo IPC.

```
struct ipc_perm {
key_t key;

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
}
```

La struttura dati associata ad un oggetto IPC è inizializzata quando l'oggetto è creato per mezzo dell'appropriata system call get.

- una volta che l'oggetto è stato creato, un programma può ottenere una copia di questa struttura dati utilizzando l'apposita syscall `ctl` e specificando un operazione di tipo `IPC_STAT`.
- alcuni elementi della struttura dati possono essere modificati per mezzo della operazione `IPC_SET`.

Modifica del campo `uid` per un segmento di memoria condivisa.

La struttura dati associata è di tipo `shmid_ds`.

```
struct shmid_ds shmds;

if (shmctl(id, IPC_STAT, &shmds) == -1){
    errExit("shmctl - IPC_STAT");
}

// Modifico l'owner ID nella mia copia locale
shmds.shm_perm.uid = newuid;

// imposto la mia copia locale come versione del kernel
// (le modifiche apportate alla mia shmds diventano globali)
```

```
if (shmctl(id, IPC_SET, &shmds) == -1){
    errExit("shmctl - IPC_SET");
}
```

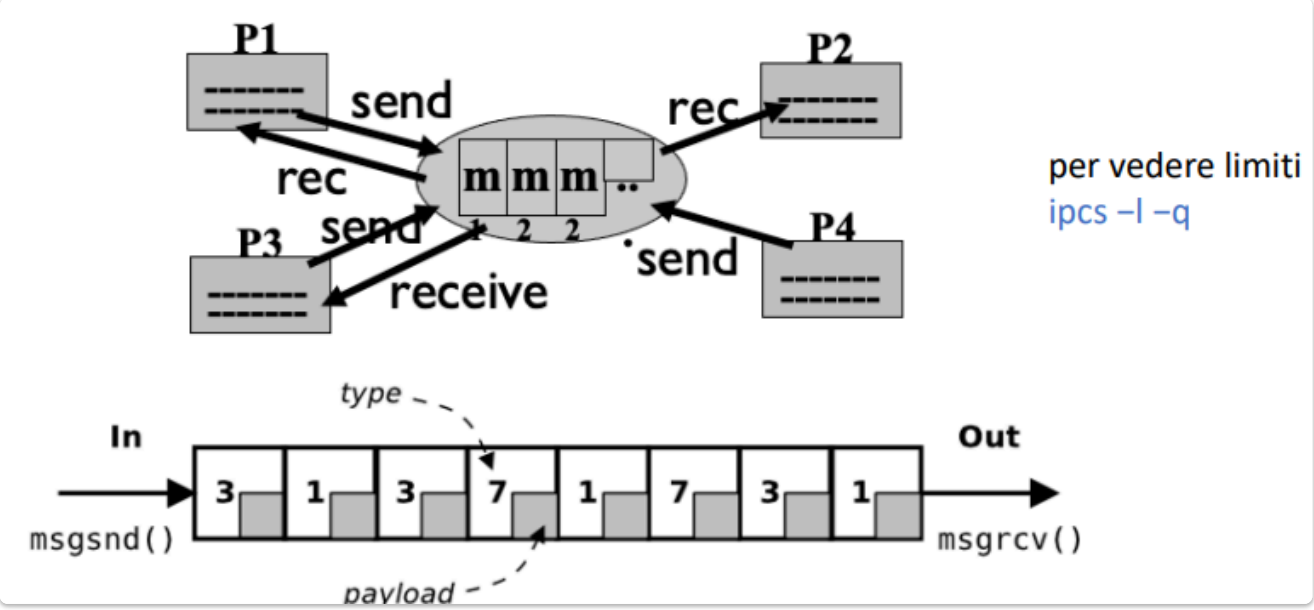
Il campo `mode` della sottostruttura `ipc_perm` contiene i permessi per l'oggetto IPC. I permessi sono inizializzati utilizzando i 9 bit più bassi specificati nei flag della syscall `get`, ma possono essere modificati successivamente utilizzando l'operazione `IPC_SET`.

Come con i file, i permessi sono divisi in tre categorie: owner (user), group e other; ed è possibile specificare i diversi permessi per ogni categoria.

Code di messaggi

Sono un canale di comunicazione su cui possono affacciarsi più processi, anche non correlati, che inviano e ricevono messaggi.

> per vedere i limiti di sistema: `ipcs -l -q`
>
> (`-l` per risposta lunga, `-q` per vedere le info solo sulle code di messaggi)



Le code di messaggi differiscono da pipe e FIFO per le seguenti caratteristiche:

- l'identificativo utilizzato per riferirsi ad una coda di messaggi è l'identificatore restituito da una chiamata a `msgget()`
- la comunicazione per mezzo di code di messaggi è 'message-oriented', cioè il lettore riceve i messaggi a blocchi interi, scritti dallo scrittore.

- non è possibile leggere porzioni di messaggi, lasciandone altre porzioni in coda, o leggere più messaggi alla volta.
- oltre a contenere dati, ogni messaggio contiene un membro di tipo intero che permette di prelevare i messaggi dalla coda in ordine fifo, oppure per tipo.

Creazione / Apertura di una coda di messaggi

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

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

L'argomento `key` è una chiave generata utilizzando:

- un numero casuale
- `IPC_PRIVATE`
- `ftok()`
  L'argomento `msgflg` è una maschera di bit che specifica i permessi da associare a una nuova coda di messaggi. Se la coda esiste già, permette di verificarne i permessi.

Parametro `msgflg`

Zero o più dei seguenti flag possono essere concatenati in ORing nel `msgflg` per controllare la `msgget()`:

- `IPC_CREAT`: se non esiste una coda con la key specificata, crea la nuova coda
- `IPC_EXCL`: se è presente anche `IPC_CREAT`, e una coda con la key specificata esiste già, restituisci un fallimento con errore `EEXIST`.

System call `msgget()`

La system call `msgget()` inizia cercando all'interno delle code di messaggi esistenti quello con la key specificata.

- se tale key corrisponde ad una coda, la `msgget()` restituisce l'identificatore di quella coda (a meno che siano stati specificati sia `IPC_CREAT` che `IPC_EXCL`, nel qual caso viene restituito l'errore `EEXIST`).

- se tale coda non esiste e `IPC_CREAT` è specificato, la `msgget()` crea una nuova coda e ne restituisce l'identificatore al chiamante.

```
// Alloco una nuova coda di messaggi tramite msgget
if ((m_id = msgget(ftok("f_name.c", 1), IPC_CREAT)) == -1){
    errExit("msgget");
}
//
if ((m_id = msgget(IPC_PRIVATE, 0644)) == -1){
    errExit("msgget");
}
```

Condivisione del valore di key

La condivisione del valore di key può avvenire in diversi modi:

1. In un file di definizioni `f_header.h`, incluso da tutti i processi che devono usare la stessa coda: `#define MYKEY 1234`
   Il processo responsabile per l'allocazione della coda eseguirà:
   `int q_id = msgget(MYKEY, IPC_CREAT | 0666);`
   Un processo che dovrà usare la coda associata a `MYKEY` eseguirà:
   `int q_id = msgget(MYKEY, 0);`
   Se la coda associata a `MYKEY` esiste, viene restituito il suo identificatore, altrimenti viene restituito -1.
2. Se la coda viene usata da un gruppo di processi fratelli, allora possono sfruttare la caratteristica del padre in comune:
   `int q_id = msgget(getppid(), 0)
3. Se la coda viene usata da processi in relazione padre-figlio si può sfruttare il fatto che il figlio eredita una copia delle variabili del padre:

```
    qid = msgget(IPC_PRIVATE, 0664);
    child_pid = fork();
    switch (child_pid){
        case -1:
            errExit("fork error.");
            break;
        case 0: // Child
            /* azioni del figlio */
            break;
        default: // Parent
            /* azioni del padre*/
```

```
            break;
    }
```

Invio e ricezione di messaggi

Le system call `msgsnd()` e `msgrcv()` eseguono le operazioni di I/O sulle code di messaggi.
In entrambe le chiamate, il primo argomento è `msqid`, un identificatore di coda di messaggi.
Il secondo argomento, `msgp` è un puntatore a struttura "programmer-defined" utilizzata per contenere il messaggio inviato o ricevuto. La struttura ha la seguente forma:

```
struct mymsg {
    long mtype; // message type
    ... mtext ... // message body
}
```

🖊 Dichiarazione di `msgsnd()`:

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

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

La system call `msgsnd()` scrive un messaggio su una coda si messaggi.
Per inviare un messaggio con la `msgsnd()` è necessario assegnare il membro `mtype` della struttura ad un valore maggiore di 0 e copiare i dadi da trasmettere nei membri della struttura.

L'argomento `msgsz` specifica il numero di bytes contenuti nel membro `mtext` della struttura. (per i limiti di sistema: `cat /proc/sys/kernel/msgmax`).

L'argomento `msgflg` è una maschera in ORing dei flag che controllano l'operazione di `msgsnd()`. È definito solo un flag:

- `IPC_NOWAIT`: consente di eseguire una *non blocking send*.
  Di norma, se una coda è piena, `msgsnd()` si blocca finché non si libera

abbastanza spazio per il messaggio che si desidera aggiungere. Invece, se è specificato questo flag, la `msgsnd()` restituisce immediatamente con l'errore `EAGAIN`.

Un esempio di utilizzo di `msgsnd()` è dunque:

```
struct q {
    long mtype;
    ... mtext
}
msgsnd(m_id, &q, sizeof(q) - sizeof(long), IPC_NOWAIT);
```

Dichiarazione di `msgrcv()`:

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

ssize_t msgrcv (int msqid, void *msgp, size_t maxmsgsz, long
msgtyp, int msgflg);

// Returns numbers of bytes copied into mtext field, -1 on error
```

La capienza del membro `mtext` del buffer `msgp` è espressa dall'argomento `maxmsgsz`. Se il corpo del messaggio da rimuovere dalla coda supera `maxmsgsz` bytes, nessun messaggio viene rimosso dalla coda e `msgrcv()` fallisce con errore `E2BIG`.

Utilizzo dell'argomento `msgtyp`

Non necessariamente i messaggi vengono letti nell'ordine con cui sono stati scritti e inviati alla coda. È possibile selezionarli utilizzando il valore contenuto nel membro `type`. Questa selezione è controllata dall'argomento `msgtyp`:

- se `msgtyp` è uguale a 0, viene prelevato il primo messaggio dalla cosa e restituito al processo chiamante
- se `msgtyp` è maggiore di 0, viene prelevato il primo messaggio in cui `mtype` è uguale al `msgtyp` e restituito al chiamante.
- se `msgtyp` è minore di 0, la coda è trattata come una coda con priorità. Viene prelevato e restituito per primo il messaggio con il minimo `mtype`

minore o uguale al valore assoluto di `msgtyp`

esempio di coda di priorità (`msgtyp` < 0):
Supponiamo di continuare ad effettuare chiamate a `msgrcv` nella forma:

```
msgrcv(id, &msg, maxmsgsz, -300, 0)
```

queste chiamate msgrcv() preleverebbero i messaggi nell'ordine: 2 (type 100), 5 (type 100), 3 (type 200), e 1 (type 300). Un'ulteriore chiamata si bloccherebbe poiché il type dell'ultimo messaggio (400) supera 300.

queue position	Message type (*mtype*)	Message body (*mtext*)
1	300	...
2	100	...
3	200	...
4	400	...
5	100	...

Utilità di `msgtyp` in grandi progetti

Specificando diversi valori per `msgtyp`, vari processi possono leggere da una coda di messaggi senza competere (racing) per leggere gli stessi messaggi.

Una tecnica utile è quella in cui ciascun processo seleziona messaggi contenenti il proprio process ID, che è quindi usato per identificare il "destinatario" per il messaggio nella coda.

Utilizzo dell'argomento `msgflg`

Costituisce una maschera di bit formata mettendo in OR zero o più flag:

- `IPC_NOWAIT`: per eseguire una ricezione non blocking, se non sono presenti messaggi da leggere, di norma la chiamata a `msgrcv` diviene bloccante in attesa di un messaggio che possa leggere. Specificando `IPC_NOWAIT` la `msgrcv()` diviene non bloccante, ritornando immediatamente con errore `ENOMSG` qualora non vi siano messaggi da leggere.
- `MSG_NOERROR`: di default, se la dimensione di `mtext` eccede lo spazio disponibile stabilito da `maxmsgsz`, la `msgrcv` fallisce. Se viene specificato `MSG_NOERROR`, la `msgrcv()` rimuove il messaggio dalla coda, ne tronca l'`mtext` a `maxmsgsz` bytes, e lo restituisice al chiamante. I dati troncati sono persi.

🖊 **Un esempio di utilizzo di `msgrcv()` è dunque:**

```
// esempio con msgtyp > 0 per inidicare il destinatario
msgrcv(m_id, &q, sizeof(q) - sizeof(long), getpid(), 0)
```

Errori di `msgsnd()` e `msgrcv()`

In caso di errore, oltre a ritornare -1, impostano `errno` secondo:

- `EACCES`: no permission to operate
  La chiamata a `msgsnd()` / `msgrcv()` è stata eseguita senza averne i permessi.
- `EIDRM`
  La coda di messaggi è stata rimossa.
- `EINTR`: the process caught a signal while sleeping
  Quando bloccato sulla `msgsnd()` per coda piena
  Quando bloccato sulla `msgrcv()` per messaggio richiesto non presente
- `ENOMEM`, `E2BIG`
  Superamento dei limiti di sistema

Operazioni di controllo sulle code di messaggi

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
// returns 0 on success, -1 on error
```

L'argomento cmd specifica l'operazione da eseguire sulla coda.

- `IPC_RMID`
  Rimuove immediatamente la coda di messaggi e la sua associata struttura dati `msqid_ds`. Tutti i messaggi presenti sulla coda vanno persi e qualsiasi processo lettore o scrittore in attesa sulla coda è immediatamente svegliato, con la `msgsnd()` o la `msgrcv()` che falliscono con errore `EIDRM`. Il terzo argomento `msgctl()` è ignorato per questa operazione.
- `IPC_STAT`
  Copia la struttura `msqid_ds` nel buffer puntato da `buf`.
- `IPC_SET`
  Aggiorna i membri della struttura `msqid_ds` associata con questa coda di messaggi, utilizzando i valori presenti nel buffer puntato da `buf`.

```
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()
}

struct ipc_perm {
    key_t key;
    uid_t uid; // Owner user ID
    gid_t gid; // Owner 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
}
```

Semafori

I semafori sono utilizzati per permettere ai processi di sincronizzare le proprie azioni: per esempio permettono di sincronizzare l'accesso a un blocco di memoria condivisa, per impedire a un processo di accedere alla memoria condivisa mentre un altro processo la sta aggiornando.

ⓘ esempio:

immaginate 10 processi che vogliono incrementare di 1 il valore di un contatore (in memoria condivisa).

in teoria, al termine dell'esecuzione di tutti e 10 i processi, ci si aspetta che la variabile venga incrementata di 10. Essendo però che è incerta la lettura (reading) di una variabile e la sua riscrittura (writing) da parte dei processi, questo può portare a diversi problemi sull'integrità del dato.

Un *semaforo* è un *intero mantenuto dal kernel* il cui valore è *sempre maggiore o uguale a zero*.
Il valore di un semaforo `s` regola l'accesso alla risorsa protetta da `s`.
Ogni processo può compiere le seguenti azioni su un semaforo:

- Inizializzare il valore di `s` ad un valore intero `a` (numero di accessi concorrenti alla risorsa): `v(s) = a`
- Usare una risorsa condivisa "protetta da `s`"
  - Se `v(s) == 0`, allora il processo si blocca e attende che `v(s) > 0`
  - Decrementa `v(s)` e usa la risorsa
- Rilasciare una risorsa dopo il suo uso
  - Incrementa `v(s)` (operazione non bloccante)
- Attendere che `v(s)` diventi 0
  - Usandolo come punto di sincronizzazione

ⓘ esempio (continua):

immaginate 10 processi che vogliono incrementare di 1 il valore di un contatore (in memoria condivisa).

se ogni processo usa un semaforo per accedere al contatore (decrementa semaforo, legge e aggiorna valore variabile e incrementa semaforo rendendo la risorsa disponibile) si eliminano possibili inconistenze
è il kernel ad assicurare tutto ciò!

Utilizzo dei semafori

header file	<sys/sem.h>
get semaphore set	semget
semaphore action	semop
data structure	semid_ds
control	semctl

I passi per utilizzare i semafori sono:

1. creazione o apertura di un set di semafori utilizzando la `semget()`
2. inizializzazione dei semafori presenti nel set, usando l'operazione `SETVAL` o `SETALL` della `semctl()`
3. esecuzione delle operazioni sui valori del semaforo, utilizzando `semop()`
   i processi che utilizzano il semaforo tipicamente usano tali operazioni per indicare l'acquisizione e il rilascio di una risorsa condivisa.
4. Quando tutti i processi hanno terminato di usare il set di semafori, rimozione del set per mezzo dell'operazione `IPC_RMID` della `semctl()`

I semafori di System V sono resti complessi dal fatto di essere allocati in gruppi detti *set di semafori*
Il numero di semafori in un set è specificato al momento della creazione del set, per mezzo della system call `semget()`. Molto spesso si utilizza un solo semaforo alla volta, ma la system call `semop()` consente di eseguire atomicamente un gruppo di operazioni su vari semafori in uno stesso set.

Creare o agganciare un set di semafori

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

int semget(key_t key, int nsems, int semflg);
// returns semaphore set indentifier on success, -1 on error
```

argomento `key`

L'argomento `key` è una chiave generata utilizzando il valore `IPC_PRIVATE` o una chiave restituita da `ftok()`
Se stiamo usando `semget()` per creare un nuovo set di semafori, allora `nsems` specifica il numero di semafori in quell'insieme, e deve essere maggiore di 0. Se stiamo usando `semget()` per ottenere l'identificatore di un set esistente, `nsems` deve essere minore o uguale alla dimensione del set (o si incorrerà nell'errore `EINVAL`).

> ⚠ **Una volta creato, non è possibile modificare il numero di semafori presenti in un dato set**

argomento `semflg`

L'argomento `semflg` è una maschera di bit che specifica i permessi da assegnare a un nuovo set di semafori o da verificare su un set esistente.

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

Zero o più flag possono essere inclusi in ORing nel `semflg` per controllare l'operazione `semget()`

- `IPC_CREAT`: se non esiste un set di semafori con la chiave specificata, crea un nuovo set
- `IPC_EXCL`: se è stato specificato anche `IPC_CREAT`, e un set di semafori con la chiave specificata esiste già, restituisci un fallimento con errore `EEXIST`

Operazioni di controllo sui semafori

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

- L'argomento `semid` è l'identificatore del set di semafori sul quale l'operazione viene eseguita.
- Per le operazioni su un singolo semafori l'argomento `semnum` identifica un semaforo interno del set. Per le altre operazioni questo argomento è ignorato, e possiamo lasciarlo a zero.
- L'argomento `cmd` specifica l'operazione

Alcune operazioni richiedono un quarto argomento di nome `arg`.
Si tratta di una union che deve essere esplicitamente definita nei nostri programmi (a meno che non sia già presente in `<sys/types.h>`, come in Mac OS X).

union semun

```
union semun {
    // value for SETVAL
    int val;
    // buffer for IPC_STAT, IPC_SET
    struct semid_ds* buf;
    // array for GETALL
    unsigned short* array;
    // Linux specific part
#if defined(__linux__)
    // buffer for IPC_INFO
    struct seminfo* __buf;
#endif
}
```

campo operazioni `cmd`

Le seguenti operazioni (`IPC_RMID` , `IPC_STAT` , `IPC_SET`) sono le stesse applicabili agli altri tipi di oggetti IPC di System V.

- `IPC_RMID` : rimuove immediatamente il set di semafori e l'associata struttura `semid_ds` . Qualsiasi processo bloccato in chiamate `semop()` in attesa su semafori è immediatamente svegliato, e `semop()` riporta l'errore `EIDRM` . L'argomento `arg` non è richiesto
- `IPC_STAT` : copia la struttura `semid_ds` associata con il set di semafori nel buffer puntato da `arg.buf` .
- `IPC_SET` : aggiorna i membri della struttura `semid_ds` associata al set di semafori utilizzando i valori nel buffer puntato da `arg.buf`

Leggere o inizializzare un semaforo

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

Le seguenti operazioni prelevano o inizializzano il valore (o i valori) di un singolo semaforo o di tutti i semafori del set.

ricorda!

La copia del valore di un semaforo richiede permessi in lettura sul semaforo, mentre l'inizializzazione del valore richiede permessi in scrittura.

- `GETVAL`: come risultato della chiamata, `semctl()` restituisce il valore del semaforo (numero) `semnum` nel set di semafori specificato da `semid`. L'argomento `arg` non è richiesto.
- `SETVAL`: il valore del semaforo `semnum` nel set riferito da `semid` è inizializzato al valore di `arg.val`
- `GETALL`: preleva i valori di tutti i semafori nel set riferito da `semid`, copiandoli nell'array puntato da `arg.array`. Il programmatore deve garantire che l'array sia abbastanza capiente. `semnum` è ovviamente ignorato.
- `SETALL`: inizializza tutti i semafori del set riferito da `semid`, usando i valori forniti nell'array puntato da `arg.array`. `semnum` è ovviamente ignorato.

Ottenere informazioni sul semaforo

Le seguenti funzioni restituiscono (attraverso il valore di ritorno della funzione) informazioni sul semaforo `semnum` del set riferito da `semid`.

> *Per tutte le operazioni, è richiesto il permesso in lettura sui semafori, e l'argomento `arg` non è richiesto.*

- `GETPID`: restituisce il PID dell'ultimo processo che ha eseguito una `semop()` su questo semaforo; questo è riferito come il valore `sempid`. Se nessun processo ha ancora eseguito una `semop()` su questo semaforo, restituisce 0.
- `GETNCNT`: restituisce il numero di processi attualmente in attesa di un incremento del valore del semaforo; questo è riferito come il valore `semncnt`.
- `GETZCNT`: restituisce il numero di processi attualmente in attesa che il valore del semaforo divenga 0; questo è riferito come il valore `semzcnt`.

struct semid_ds

```
struct semid_ds {
    // owenership and permissions
    struct ipc_perm sem_perm;
    // last semop time
```

```
    time_t sem_otime;
    // last change time
    time_t sem_ctime;
    // number of semaphores in set
    unsigned short sem_nsems;
}
```

Ogni set di semafori ha associata una struttura `semid_ds`.
I membri della struttura `semid_ds` sono implicitamente aggiornati da varie system call sul semaforo, e alcuni membri della struttura `sem_perm` possono essere aggiornati esplicitamente con l'operazione `semctl()` tramite `IPC_SET`.

- *sem_perm* quando il set di semafori è creato, i membri di questa struttura sono inizializzati. I membri `uid`, `gid` e `mode` possono essere aggiornati con l'operazione `IPC_SET`.
- *sem_otime* questo membro è settato a 0 alla creazione del set di semafori, ed è aggiornato all'ora corrente ad ogni `semop()` che va a buon fine, o quando il valore del semaforo è modificato in seguito all'operazione `SEM_UNDO`.
- *sem_ctime* questo membro è impostato all'ora corrente al momento della creazione del semaforo, ed in seguito ad ogni operazione `IPC_SET`, `SETALL` o `SETVAL`.
- *sem_nsems* membro inizializzato al momento della creazione del set di semafori: contiene il numero di semafori nel set.

Operazioni sui semafori

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

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

La system call `semop()` esegue una o più operazioni sui semafori nel set indicato da `semid`.

- *sops* è un puntatore ad un array che contiene le operazioni da eseguirsi
- *nsops* è la dimensione dell'array (che deve contenere almeno un elemento)

specifica dell'operazione con `sembuf`

```
struct sembuf{
    unsigned short sem_num; // numero semaforo
    short sem_op; // operazione da eseguire
    short sem_flg; // flags operazione (IPC_NOWAIT e SEM_UNDO)
}
```

Le operazioni sono eseguite atomicamente e nell'ordine in cui compaiono nell'array. Gli elementi dell'array `sops` richiesto da `semop()` sono strutture `sembuf` con le seguenti caratteristiche:

sem_num
Il membro `sem_num` identifica, all'interno del set, il semaforo sul quale si intende effettuare l'operazione.

sem_op
Il membro `sem_op` specifica l'operazione da eseguire:

- se `sem_op` è maggiore di 0, il valore di `sem_op` è aggiunto al valore del semaforo.
  Quindi altri processi in attesa di diminuire il valore del semaforo possono essere svegliati per eseguire le proprie operazioni.
- se `sem_op` è uguale a 0, il valore del semaforo è testato per vedere se attualmente è uguale a 0. Se lo è, l'operazione è completata immediatamente, diversamente la `semop()` si blocca finché il valore del semaforo diviene 0.
- se `sem_op` è minore di 0, decrementa il valore del semaforo del valore specificato da `sem_op`. Se il valore corrente del semaforo è maggiore o uguale al valore assoluto specificato da `sem_op`, l'operazione è completata immediatamente.
  Diversamente, `semop()` si blocca finché il valore del semaforo è stato aumentato tanto da permettere che l'operazione venga eseguita (senza produrre un valore negativo).

⚠ **In tutti e 3 i casi, il processo chiamante deve avere diritti di scrittura sul semaforo.**

Interpretazione delle operazioni su un semaforo

L'aumento del valore di un semaforo corrisponde a rendere disponibile una risorsa così che altri processi possano utilizzarla.

La diminuzione del valore del semaforo corrisponde a riservare la risorsa per un uso esclusivo da parte di questo processo.
Il tentativo di ridurre il valore di un semaforo può restare bloccato se il valore del semaforo è troppo basso, garantendo uno strumento per gestire un uso esclusivo della risorsa in questione.

Cosa accade ad un processo bloccato?

Quando una system call `semop()` si blocca, il processo resta bloccato finché:

- Un altro processo modifica il valore del semaforo così che la richiesta possa procedere.
- Un segnale interrompe la system call `semop()`. In questo caso la `semop()` fallisce con l'errore `EINTR`.
- Un altro processo cancella il semaforo identificato da `semid`. In questo caso, la `semop()` fallisce con l'errore `EIDRM`.

Non-blocking `semop()`

È possibile prevenire il blocco della `semop()` nell'esecuzione di un'operazione su un dato semaforo specificando il flag `IPC_NOWAIT` nel membro `sem_flg` della struttura `sembuf`. In questo caso, invece di essere bloccata, la `semop()` fallisce ritornando -1 e `errno` conterrà l'errore `EAGAIN`.

Operazioni su molteplici semafori

È possibile eseguire una `semop()` per compiere operazioni su molteplici semafori in uno stesso set. Questo gruppo di operazioni è eseguito atomicamente: o la `semop()` esegue tutte le operazioni, o si blocca fino a quando non diventa possibile eseguirle tutte simultaneamente.

consideriamo un esempio:

l'uso di `semop()` per eseguire operazioni su tre semafori in un set.
Le operazioni sui semafori 0 e 2 possono non essere in grado di procedere

immediatamente, a seconda dei valori correnti dei semafori.
Se l'operazione sul semaforo 0 non può essere eseguita immediatamente, allora nessuna delle operazioni richieste viene eseguita, e la `semop()` si blocca.
Se l'operazione sul semaforo 0 può essere eseguita immediatamente, ma non l'operazione sul semaforo 2, allora se è stato specificato il flag `IPC_NOWAIT` sul semaforo 2, nessuna delle operazioni è eseguita, e la `semop()` restituisce immediatamente con l'errore `EAGAIN`.

Altro esempio:

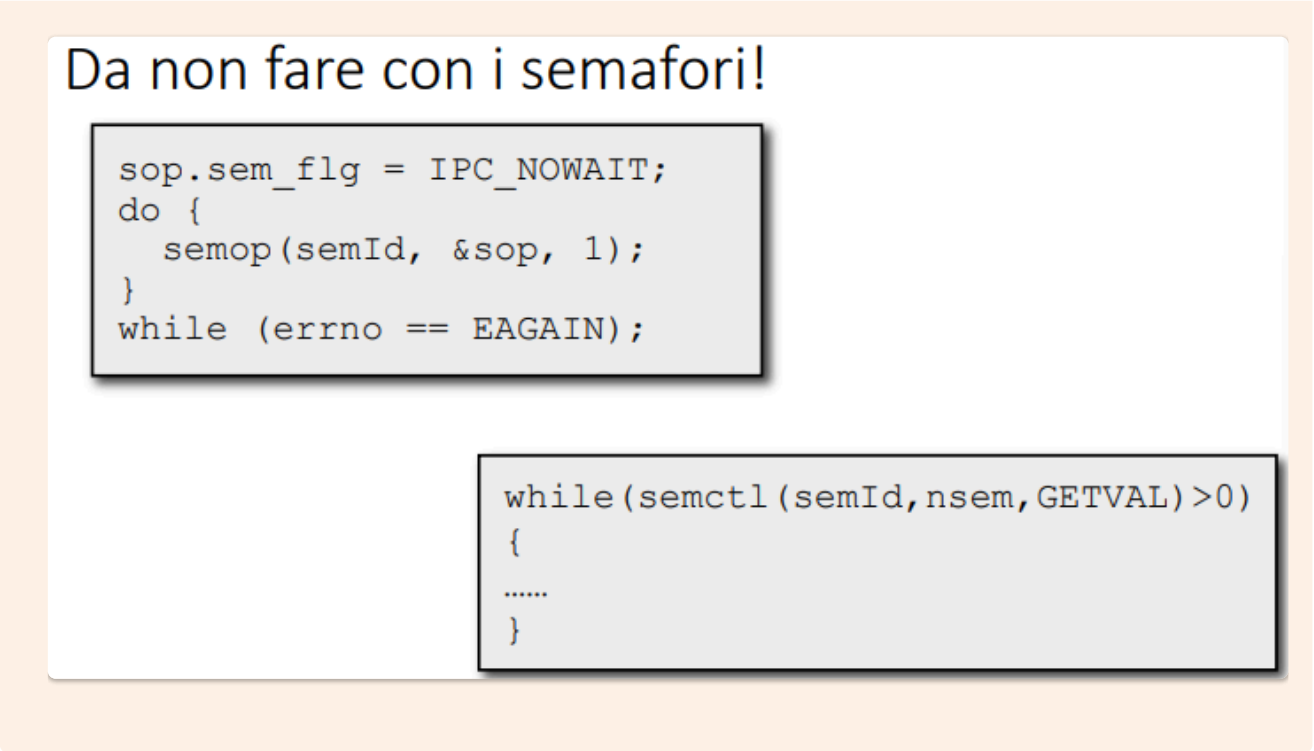
```
struct sembuf sops[3];

sops[0].sem_num = 0;
sops[0].sem_op = -1; // DECREMENTO di 1 il semaforo 0
sops[0].sem_flg = 0;

sops[1].sem_num = 1;
sops[1].sem_op = 2; // INCREMENTO di 2 il semaforo 1 */
sops[1].sem_flg = 0;

sops[2].sem_num = 2;
sops[2].sem_op = 0; // ATTESA che il semaforo 2 valga 0 */
sops[2].sem_flg = IPC_NOWAIT; /* operazione NON SOSPENSIVA: se l'operazione
non può essere effettuata, NON attendere */

if (semop(semid, sops, 3) == -1) {
   if (errno == EAGAIN) /* il semaforo 2 si sarebbe bloccato; invece grazie
a IPC_NOWAIT ha restituito EAGAIN */
   printf("Operation would have blocked\n");
else
   errExit("semop"); // si è verificato qualche altro errore
```

Semafori e segnali

Quando un processo è bloccato in una `semop()` e riceve un segnale non mascherato, il corrispondente handler è eseguito, la `semop()` si blocca, ritorna -1 e setta `errno` a `EINTR`.

Anche se il flag `SA_RESTART` è stato precedentemente settato nell'handler del segnale `sigaction()`, una `semop()` che è stata interrotta, terminerà sempre con fallimento ed `errno` a `EINTR`.

⚠ **DA NON FARE CON I SEMAFORI:**



semafori con soglia temporale di attesa

È possibile anche sbloccare un processo in attesa su un semaforo dopo un certo intervallo di tempo anche se la risorsa non è disponibile. Allo sblocco, la `semtimedop` restituisce -1 con `errno` a `EAGAIN` se il timeout è scaduto:

```
int semtimedop (int semid, struct sembuf *sops, size_t nsops,
const struct timespec *timeout);
```

con `timeout` dichiarato all'occorrenza tramite:

```
struct timespec timeout;
timeout.tv_sec = 5; // seconds
timeout.tv_nsec = 5000; // nanoseconds
```

Semafori binari

La API per i semafori di System V semaphores è complessa, perché il valore dei semafori può essere modificato di quantità arbitrarie, i semafori sono allocati in set di semafori, e le operazioni sono eseguite su set di semafori.
Queste caratteristiche forniscono funzionalità più estese di quelle tipicamente necessarie, quindi è utile implementare protocolli più semplici.

Un protocollo normalmente utilizzato è quello dei *semafori binari*.

Un semaforo binario ha due valori: *available* (libero) e *reserved* (in uso).
Sui semafori binari sono definite due operazioni:

- *Reserve* (wait, o P): Tenta di riservare questo semaforo per uso esclusivo. Se il semaforo è già stato riservato da un altro processo, l'operazione si blocca fino a quando il semaforo è rilasciato.
- *Release* (signal, o V): Libera un semaforo correntemente riservato, così che possa essere riservato da un altro processo.

Un modo largamente usato per rappresentare questi stati è utilizzare il valore 1 per indicare free e il valore 0 per riservato, con le operazioni reserve e release: l'una decrementa di uno il valore del semaforo, l'altra lo incrementa di uno.
Tutte le funzioni in questa implementazione hanno due argomenti, che identificano un set di semafori e il numero di un semaforo all'interno di quel set

```
int initSemAvailable (int semId, int semNum);
int initSemInuse (int semId, int semNum);
int reserveSem (int semId, int semNum);
int releaseSem (int semId, int semNum);
```

Possibile implementazione

```
// initialise semaphore to 1 (available)
int initSemAvailable (int semId, int semNum){
    union semun arg;
    arg.val = 1;
    return semctl(semId, semNum, SETVAL, arg);
}
// initialise semaphore to 0 (in use)
int initSemInUse (int semId, int semNum){
    union semun arg;
    arg.val = 0;
    return semctl(semId, semNum, SETVAL, arg);
}
// reserve semaphore
int reserveSem (int semId, int semNum){
    struct sembuf sops;
    sops.sem_num = semNnum;
    sops.sem_op = -1;
    sops.sem_flg = 0;
    return semop(semId, &sops, 1);
```

```
}
// release semaphore
int releaseSem (int semId, int semNum){
    struct sembuf sops;
    sops.sem_num = semNnum;
    sops.sem_op = 1;
    sops.sem_flg = 0;
    return semop(semId, &sops, 1);
}
```

Memoria Condivisa

La memoria condivisa (shared memory, SM) *consente a due o più processi di condividere la stessa regione* (tipicamente detta segmento) *di memoria fisica*. Um processo copia i dati all'interno della memoria condivisa; quei dati divengono immediatamente disponibili a tutti gli altri processi che condividono lo stesso segmento.

Si tratta di uno strumento che fornisce una IPC veloce in confronto a tecniche come pipe o code di messaggi, in cui il processo mittente copia i dati da un buffer dello spazio utente nella memoria, e in cui il ricevente effettua una copia nella direzione inversa.

Poiché l'utilizzo della SM non é mediato dal kernel, tipicamente é necessario predisporre qualche metodo di sincronizzazione, per impedire ai processi di accedere simultaneamente alla memoria condivisa.

Spazi di indirizzamento

Di norma ogni processo possiede uno spazio di indirizzamento logico separato dagli altri processi. Un segmento di memoria condivisa può essere invece letto e/o scritto da due o più processi, e permette quindi un rapido scambio di informazioni.

Ogni processo usa quindi il segmento di SM come se fosse una normale porzione del proprio spazio di indirizzamento logico, che però é fisicamente in comune a più processi.

Un figlio creato con una `fork()` eredita i segmenti di SM a disposizione del genitore. Quindi la SM fornisce uno strumento semplice per l'IPC fra genitore e figli.

Durante una `exec()`, tutti i segmenti attaccati sono staccati (detached), ma non distrutti. I segmenti sono anche automaticamente staccati al momento della terminazione dei processi.

Utilizzo della memoria condivisa

header file	<sys/shm.h>
get segment	shmget
attach segment	shmat
detach segment	shmdt
data structure	shmid_ds
control	shmctl

shmget

La chiamata `shmget()` é usata per creare un nuovo segmento di SM o per ottenere l'identificatore di un segmento esistente.

> *(i.e. un segmento creato da un altro processo).*

shmat

La chiamata `shmat()` é usata per attaccare il segmento di SM, rendendolo parte della memoria virtuale del processo chiamante. La memoria condivisa può essere trattata come qualsiasi altra porzione di memoria indirizzabile dall'interno del programma. Al fine di riferirsi alla memoria condivisa, il programma usa il valore `addr` restituito dalla chiamata `shmat()` che é un puntatore all'inizio del segmento di SM nello spazio di indirizzi virtuale del processo.

shmdt

La chiamata `shmdt()` é usata per staccare il segmento di SM. Dopo tale chiamata, il processo non può più fare riferimento alla SM.

> ⚠ **Tale passo risulta spesso "opzionale"**
>
> in quanto avviene automaticamente alla terminazione di ogni processo segmenti di SM attaccati.

shmctl

La chiamata `shmctl()` é usata per cancellare il segmento di SM. Il segmento sarà effettivamente distrutto solo dopo che tutti i processi correntemente attaccati lo avranno staccato. Un solo processo effettua la cancellazione.

Creazione di una shared memory

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

int shmget(ket_t key, size_t size, int shmflg);
// returns SM segment identifier on success, or -1 on error
```

key

L'argomento `key` é una chiave tipicamente generata usando il valore `IPC_PRIVATE` o una key restituita da `ftok()`.

size

Quando utilizziamo la `shmget()` per creare un nuovo segmento di SM, `size` specifica un intero positivo che indica la dimensione del segmento, espressa in bytes.
Il kernel alloca la SM in multipli della dimensione delle pagine di sistema (*system pages*), quindi in pratica size viene arrotondata al multiplo successivo della dimensione della pagina.

Se stiamo utilizzando `shmget()` per ottenere l'identificatore di un segmento esistente, size é un argomento privo di effetto sul segmento, ma deve essere minore o uguale alla dimensione del segmento.

shmflg

L'argomento `smhflg` svolge le stesse operazioni comuni alle altre chiamate IPC `get`, *specificando i permessi* da associare al nuovo segmento, o da verificare su un segmento esistente. Inoltre, zero o più fra i seguenti flag possono essere concatenati in ORing:

- `IPC_CREAT`: se non esiste un segmento con la key specificata, crea un nuovo segmento
- `IPC_EXCL`: se é stato specificato `IPC_CREAT`, e un segmento con la `key` specificata esiste già, fallisce con `errno` a `EEXIST`.

Area condivisa e tipi

La `shmget()` funziona in modo simile alla `malloc`, con la differenza che l'area allocata è accessibile a più processi. Si può quindi prelevare spazio facendo riferimento a diversi tipi di dati:

- Tipi fondamentali: `shmget(..., sizeof(int), ...);`
- Array: `shmget(..., sizeof(char) * N, ...);`
- Strutture: `shmget(..., sizeof(struct libro), ...);`
- Array di tipi derivati: `shmget(..., sizeof(struct dato) * N, ...);`

Attach della memoria condivisa

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

shmid

La system call `shmat()` attacca un'area di memoria identificata da `shmid` allo spazio di indirizzamento del processo.

shmaddr

L'argomento `shmaddr` e l'impostazione del `SHM_RND` bit nella bit-mask `shmflg` controllano il modo in cui il segmento é attaccato:

- se `shmaddr` é `NULL`, allora il segmento é attaccato all'indirizzo appropriato dal kernel. Questo é il modo migliore per attaccare un segmento.

> ⚠ **Specificare un valore non-`NULL` per `shmaddr` non é raccomandabile:**
>
> 1. Riduce infatti la portabilità di un applicazione. Un indirizzo valido in una implementazione UNIX può non essere valido in un'altra.
> 2. Un tentativo di attaccare un segmento di SM ad un particolare indirizzo fallirà se quell'indirizzo é già utilizzato.

`shmat()` restituisce l'indirizzo al quale il segmento di SM é attaccato.
Questo valore può essere trattato come un normale puntatore C, il segmento può essere trattato come qualsiasi altra parte della memoria virtuale del processo.

Tipicamente, assegnamo il valore di ritorno di `shmat()` ad un puntatore a qualche struttura definita nel programma, al fine di imporre quella struttura sul segmento:

```
struct shmseg *shmp;

shmp = (struct shmseg *)shmat(shmid, NULL, 0);
if (shmp == (void *)-1) { errExit("shmat error"); }
```

shmflg
Possibilità di mettere alcune flag in ORing:

- Per attaccare un segmento di SM per un accesso read-only, specifichiamo il flag `SHM_RDONLY` nel `shmflg`. Così facendo, tentativi di aggiornare i contenuti di un segmento disponibile solo in lettura produrranno un segmentation fault comunicato tramite segnale `SIGSEGV`. Se `SHM_RDONLY` non é specificato, l'operazione di attach di default garantisce permessi di lettura e scrittura della memoria.
- `SHM_REMAP`: Replace any existing mapping at `shmadrr`.
- `SHM_RND`: round `shmaddr` down to multiple of `SHMLBA` bytes.

Detach di una memoria condivisa

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

Quando un processo non accede più ad un segmento di SM, può chiamare la system call `shmdt()` per staccare il segmento dal proprio spazio di indirizzi virtuale.

L'argomento `shmaddr` identifica il segmento da staccare. Dovrebbe essere un valore restituito da una precedente chiamata `shmat()`.

La `shmdt()` sgancia l'area di memoria condivisa dallo spazio degli indirizzi del processo chiamante. Lo sganciamento non equivale alla cancellazione dell'area di memoria, che deve essere eseguita per mezzo della `shmctl()` con il comando `IPC_RMID`.

ⓘ Esempio di utilizzo di una shared memory

```c
int main(int argc, char** argv) {
   int shmid_1, shmid_2, return_val;
   char *stringa_1, *stringa_2;
   char msg[] = "ciao a tutti!";

   shmid_1 = shmget(MYKEY, sizeof(char)*SHMSZ, IPC_CREAT|0666);
   stringa_1 = (char *)shmat(shmid_1,NULL,0);

   snprintf(stringa_1, sizeof(msg), "%s", msg);
   return_val = shmdt(stringa_1);

   // un altro processo che si attacchi alla stessa area
   // di memoria condivisa potrà leggerne il contenuto
   shmid_2 = shmget(MYKEY, sizeof(char)*SHMSZ, 0);

   stringa_2 = (char *)shmat(shmid_2,NULL,0);
   printf("%s\n",stringa_2);

   shmctl(shmid_2, IPC_RMID,0);
   exit(EXIT_SUCCESS);
}
```

Operazioni di controllo su una memoria condivisa

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

La system call `shmctl()` esegue un insieme di operazioni di controllo sul segmento di SM identificato da `shmid`.

L'argomento `cmd` specifica l'operazione da eseguire.

L'argomento `buf` é richiesto dalle operazioni `IPC_STAT` e `IPC_SET`, e dovrebbe essere lasciato a `NULL` per le altre.

operazioni

- `IPC_RMID`: marca il segmento di SM e l'associata struttura `shmid_ds` per la cancellazione. Se nessun processo ha il segmento attaccato, la cancellazione é immediata; diversamente, il segmento é rimosso dopo che tutti i processi lo hanno staccato. Tale operazione é analoga all'unlinking dei file una volta che li abbiamo aperti (vedi `man 2 unlink`).

  🔥 esempio di uso di `IPC_RMID`

In alcune applicazioni possiamo assicuraci che un segmento sia rimosso al momento della terminazione dell'applicazione marcandolo per la cancellazione immediatamente dopo che tutti i processi lo hanno attaccato al proprio spazio di indirizzi con la `shmat()`.

- `IPC_STAT`: copia la struttura `shmid_ds` associata a questo segmento nel buffer puntato da `buf`.
- `IPC_SET`: aggiorna i membri della struttura `shmid_ds` associata a questo segmento con il buffer puntato da `buf`.

Struttura dati di `shmid_ds`

```
struct shmid_ds {
struct ipc_perm shm_perm; /*Ownership e 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 last creator */
pid_t shm_lpid; /* PID of last shmat() / shmdt() */
shmatt_t shm_nattch; /* Number of currently attached processes */
}
```

⚠ **NON ESISTE UNA FUNZIONE SIMILE A `realloc` PER LA MEMORIA CONDIVISA**

Deve essere "fatta a mano":

- bloccare (su un semaforo) tutti i processi prima di accedere: `semop(...)`
- creare un segmento di memoria condivisa più grande: `shmget(...)`
- collegarsi alla nuova memoria condivisa: `shmat(...)`
- copiare i vecchi dati nel nuovo segmento: `bcopy(..)`
- rimuovere il vecchio segmento: `shmctl(..., IPC_RMID)`
- comunicare l'ID del nuovo segmento di SM a tutti i processi coinvolti