LabSO 2025

Laboratorio Sistemi Operativi - A.A. 2024-2025

Michele Grisafi - michele.grisafi@unitn.it

Segnali

Segnali in Unix

Ci sono vari eventi che possono avvenire in maniera asincrona al normale flusso di un programma, alcuni dei quali in maniera inaspettata e non predicibile. Per esempio, durante l'esecuzione di un programma ci può essere una richiesta di terminazione o di sospensione da parte di un utente, la terminazione di un processo figlio o un errore generico.

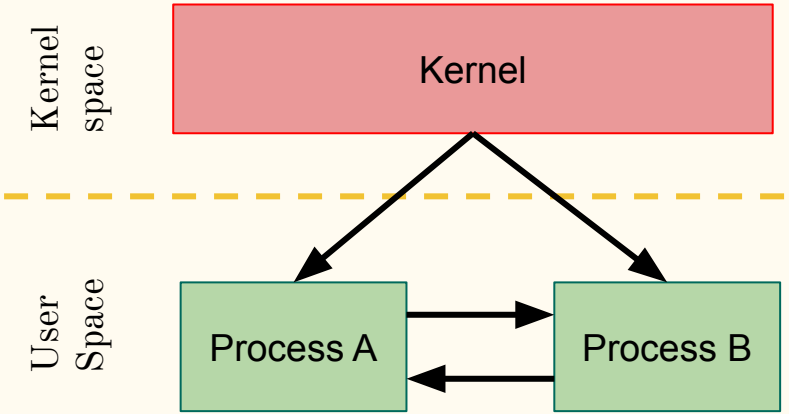
Unix prevede la gestione di questi eventi attraverso i **segnali**: quando il sistema operativo si accorge di un certo evento, genera un segnale da mandare al processo interessato il quale potrà *decidere* (nella maggior parte dei casi) come comportarsi.

Segnali in Unix

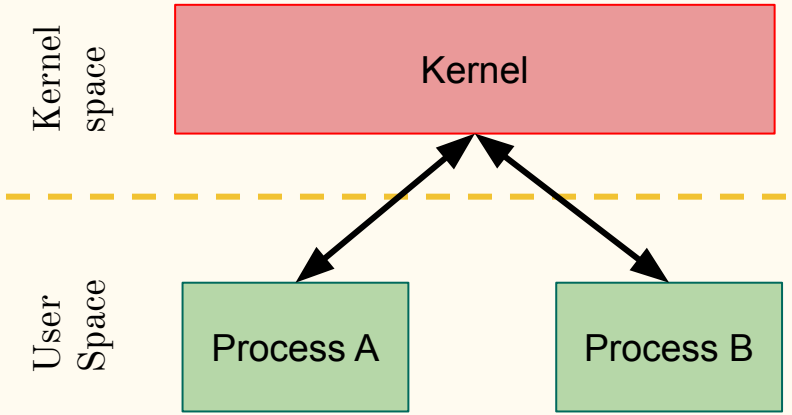
Il numero dei segnali disponibili cambia a seconda del sistema operativo, con Linux che ne definisce 32. Ad ogni segnale corrisponde sia un valore numerico che un'etichetta mnemonica (definita nella libraria "*signal.h*") nel formato `SIG`XXX. Alcuni esempi:

`SIGALRM` (alarm clock)
`SIGCHLD` (child terminated)
`SIGCONT` (continue, if stopped)
`SIGINT` (terminal interrupt, CTRL + C)
`SIGKILL` (kill process)
`SIGQUIT` (terminal quit)
`SIGSTOP` (stop)
`SIGTERM` (termination)
`SIGUSR1` (user signal)
`SIGUSR2` (user signal)

Visione concettuale



Vedete qualche problema?



Il kernel media tutti i segnali!

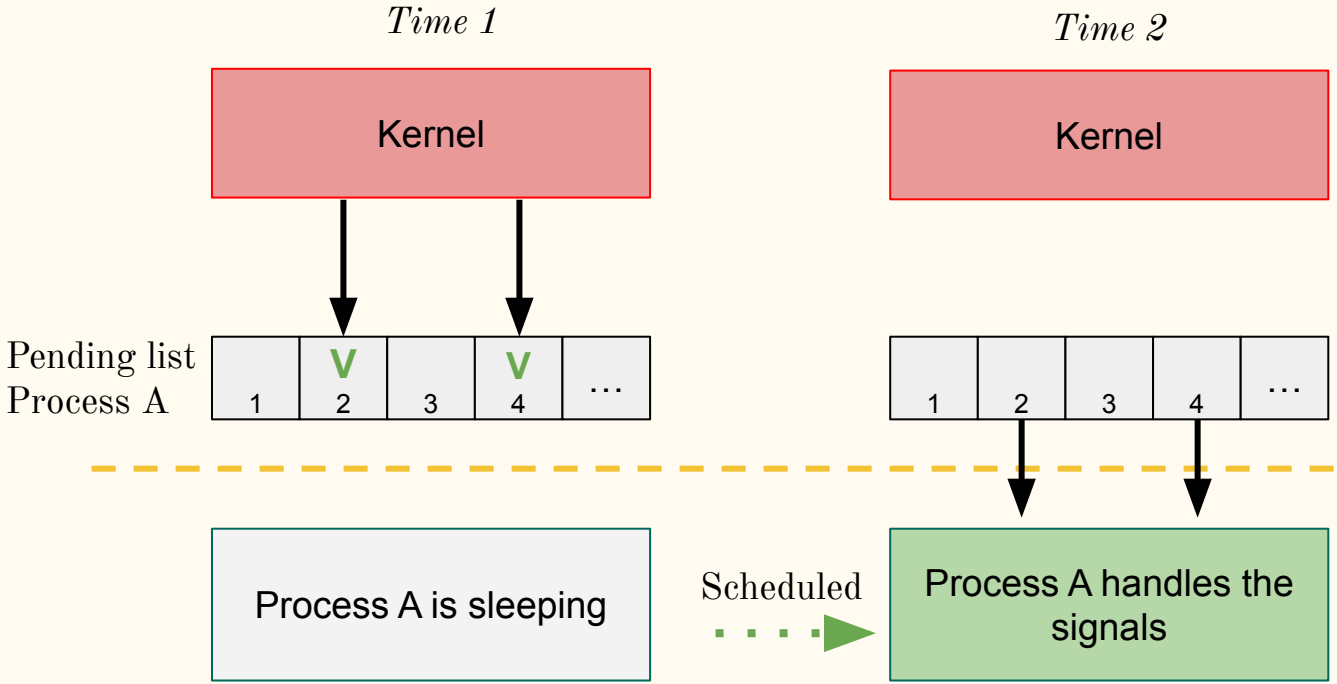
Segnali in Unix

Per ogni processo, all'interno della process table, vengono mantenute due liste:

- **Pending signals**: segnali emessi dal kernel e che il processo deve ancora gestire.
- **Blocked signals**: segnali che non devono essere comunicati al processo.
  Chiamata anche con il termine **signal mask**, maschera dei segnali.

Ad ogni schedulazione del processo le due liste vengono controllate per consentire al processo di reagire nella maniera più adeguata.

Visione concettuale dei segnali



Gestione dei segnali

I segnali sono anche detti "*software interrupts*" perchè sono, a tutti gli effetti, delle interruzioni del normale flusso del processo generate dal sistema operativo (invece che dall'hardware, come per gli *hardware interrupts*).

Come per gli interrupts, il programma può decidere come gestire l'arrivo di un segnale (presente nella lista *pending*):

- Eseguendo l'azione default.
- Ignorandolo (non sempre possibile) → programma prosegue normalmente.
- Eseguendo un handler personalizzato → programma si interrompe.

NB: nella pratica, il programma comunica al kernel come vuole che il segnale venga gestito, ed è poi il kernel che richiamerà la funzione adeguata del programma.

Default handler

Ogni segnale ha un suo handler di default che tipicamente può:

- Ignorare il segnale
- Terminare il processo
- Continuare l'esecuzione (se il processo era in stop)
- Interrompere il processo

Ogni processo può sostituire il gestore di default con una funzione "custom" (**a parte per SIGKILL e SIGSTOP**) e comportarsi di conseguenza. La sostituzione avviene tramite la system call `signal()` (definita in "*signal.h*").

signal()

NB: deprecato ormai!

```
sighandler_t signal(int signum, sighandler_t handler);
```

Imposta un nuovo signal handler `handler` per il segnale `signum`. Restituisce il signal handler precedente. Quello nuovo può essere:

- `SIG_DFL`: handler di default
- `SIG_IGN`: ignora il segnale
- `typedef void (*sighandler_t)(int)`: custom handler.

```
#include <signal.h> <stdio.h> <stdlib.h>
void main(){
    signal(SIGINT,SIG_IGN);    //Ignore signal
    signal(SIGCHLD,SIG_DFL);   //Use default handler
}
```

Custom handler

Un custom handler deve essere una funzione di tipo `void` che accetta come argomento un `int` rappresentante il segnale catturato. Questo consente allo stesso handlers di gestire segnali diversi.

```
#include <signal.h> <stdio.h>                                    //param.c

void myHandler(int sigNum){
    if(sigNum == SIGINT) printf("CTRL+C\n");
    else if(sigNum == SIGTSTP) printf("CTRL+Z\n");
}

signal(SIGINT,myHandler);
signal(SIGTSTP,myHandler);
```

Esempio:

```
$ ./sig.out
$ <CTRL+Z>
```

```c
#include <signal.h>    //sigCST.c
#include <stdio.h>

void myHandler(int sigNum){
    printf("CTRL+Z\n");
    exit(2);
}

int main(void){
    signal(SIGTSTP,myHandler);
    while(1);
}
```

```c
#include <signal.h>    //sigDFL.c
int main(){
    signal(SIGTSTP,SIG_DFL);
    while(1);
}
```

```c
#include <signal.h>    //sigIGN.c
int main(){
    signal(SIGTSTP,SIG_IGN);
    while(1);
}
```

signal() return

signal() restituisce un riferimento all'handler che era precedentemente assegnato al segnale:

- `NULL`: handler precedente era l'handler di default
- `1`: l'handler precedente era `SIG_IGN`
- `<address>`: l'handler precedente era `*(address)`

```
#include <signal.h> <stdio.h>                                   //return.c
void myHandler(int sigNum){}
int main(){
    printf("DFL: %p\n",signal(SIGINT,SIG_IGN));
    printf("IGN: %p\n",signal(SIGINT,myHandler));
    printf("Custom: %p == %p\n",signal(SIGINT,SIG_DFL),myHandler);
}
```

Alcuni segnali

SIGXXX	**description**	**default**
SIGALRM	(alarm clock)	quit
SIGCHLD	(child terminated)	ignore
SIGCONT	(continue, if stopped)	ignore
SIGINT	(terminal interrupt, CTRL + C)	quit
SIGKILL	(kill process)	quit
SIGSYS	(bad argument to syscall)	quit with dump
SIGTERM	(software termination)	quit
SIGUSR1/2	(user signal 1/2)	quit
SIGSTOP	(stopped)	suspend
SIGTSTP	(terminal stop, CTRL + Z)	suspend

Esempio: SIGCHLD

```
#include <signal.h> <stdio.h> <sys/wait.h> <unistd.h>   //child.c
int childStatus = 0; pid_t child_pid = 0;
void myHandler(int sigNum){
    printf("Child exit! Sig %d - code %d - pid %ld!\n",
                          sigNum,WEXITSTATUS(childStatus),child_pid);
}
int main(){
    signal(SIGCHLD,myHandler);
    int child = fork();
    if(!child){
        sleep(2); return 0;
    }
    child_pid = wait(&childStatus);
    printf("Child status after wait %d - pid %ld\n",
                               WEXITSTATUS(childStatus),child_pid);
}
```

Invio di segnali

Inviare i segnali: kill()

`int kill(pid_t pid, int sig);` NB: ordine degli argomenti!

`$ kill -<signo> <pid_t>`

Invia un segnale ad uno o più processi a seconda dell'argomento *pid:*

- `pid > 0`: segnale al processo con `PID=pid`
- `pid = 0`: segnale ad ogni processo dello stesso gruppo
- `pid = -1`: segnale ad ogni processo possibile (stesso UID/RUID)
- `pid < -1`: segnale ad ogni processo del gruppo `|pid|`

Restituisce 0 se il segnale viene inviato, -1 in caso di errore.

Ogni tipo di segnale può essere inviato, non deve essere necessariamente un segnale corrispondente ad un evento effettivamente avvenuto!

```c
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>
//kill.c
void myHandler(int sigNum){printf("[%d]ALARM!\n",getpid());}
int main(void){
    signal(SIGALRM,myHandler);
    int child = fork();
    if (!child) while(1); // block the child
    printf("[%d]sending alarm to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGALRM); // send ALARM, child's handler reacts
    printf("[%d]sending SIGTERM to %d in 3 s\n",getpid(),child);
    sleep(3);
    kill(child,SIGTERM); // send TERM: default is to terminate
    while(wait(NULL)>0);
}
```

Kill da bash

kill è anche un programma in bash che accetta come primo argomento il tipo di segnale (`kill -l` per la lista) e come secondo argomento il PID del processo.

```c
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //bash.c
void myHandler(int sigNum){
    printf("[%d]ALARM!\n",getpid());
    exit(0);
}
int main(){
    signal(SIGALRM,myHandler);
    printf("I am %d\n",getpid());
    while(1);
}
```

```
$ gcc bash.c -o bash.out
$ ./bash.out
# On new window/terminal
$ kill -14 <PID>
```

Kill per verificare esistenza processo

Inviando un segnale 0 ad un processo, kill può controllare se un processo “esiste”. Nessun segnale verrà inviato, ma verrà controllato se il processo specificato può essere contattato.

`kill(<pid>,0)` restituirà -1 se il processo non esiste, 0 altrimenti.

NB: se un processo è uno zombie, kill restituirà comunque 0!

```c
#include <signal.h> <stdio.h> <stdlib.h> <sys/wait.h> <unistd.h>   //killExist.c
int main(){
  pid_t pids[10];
  for(int i = 0; i<10;i++){
    pids[i] = fork();
    if(pids[i] == 0){
      if(i%2==0)
        return 0;
      else
        while(1) pause();
    }
  }
  sleep(1);
  //while(waitpid(-1, NULL, WNOHANG) > 0); //We use WNOHANG so that we don't wait for those children still active
  int count = 0;
  for(int i = 0; i<10; i++) count += kill(pids[i], 0);
  printf("Count is %d\n", count); //0 if waitpid is commented out, -5 otherwise
  return 0;
}
```

Set up an alarm: alarm()

`unsigned int alarm(unsigned int seconds);`

Genera un segnale SIGALRM per il processo corrente dopo un lasso di tempo specificato in secondi. Restituisce i secondi rimanenti all'alarm precedente.

```
#include <signal.h> <stdio.h> <stdlib.h> <unistd.h>  //alarm.c
short cnt = 0;
void myHandler(int sigNum){printf("ALARM!\n"); cnt++;}
int main(){
    signal(SIGALRM,myHandler);
    alarm(0); //Clear any pending alarm
    alarm(5); //Set alarm in 5 seconds
    //Set new alarm (cancelling previous one)
    printf("Seconds remaining to previous alarm %d\n",alarm(2));
    while(cnt<1);
}
```

Bloccare i segnali

Bloccare i segnali

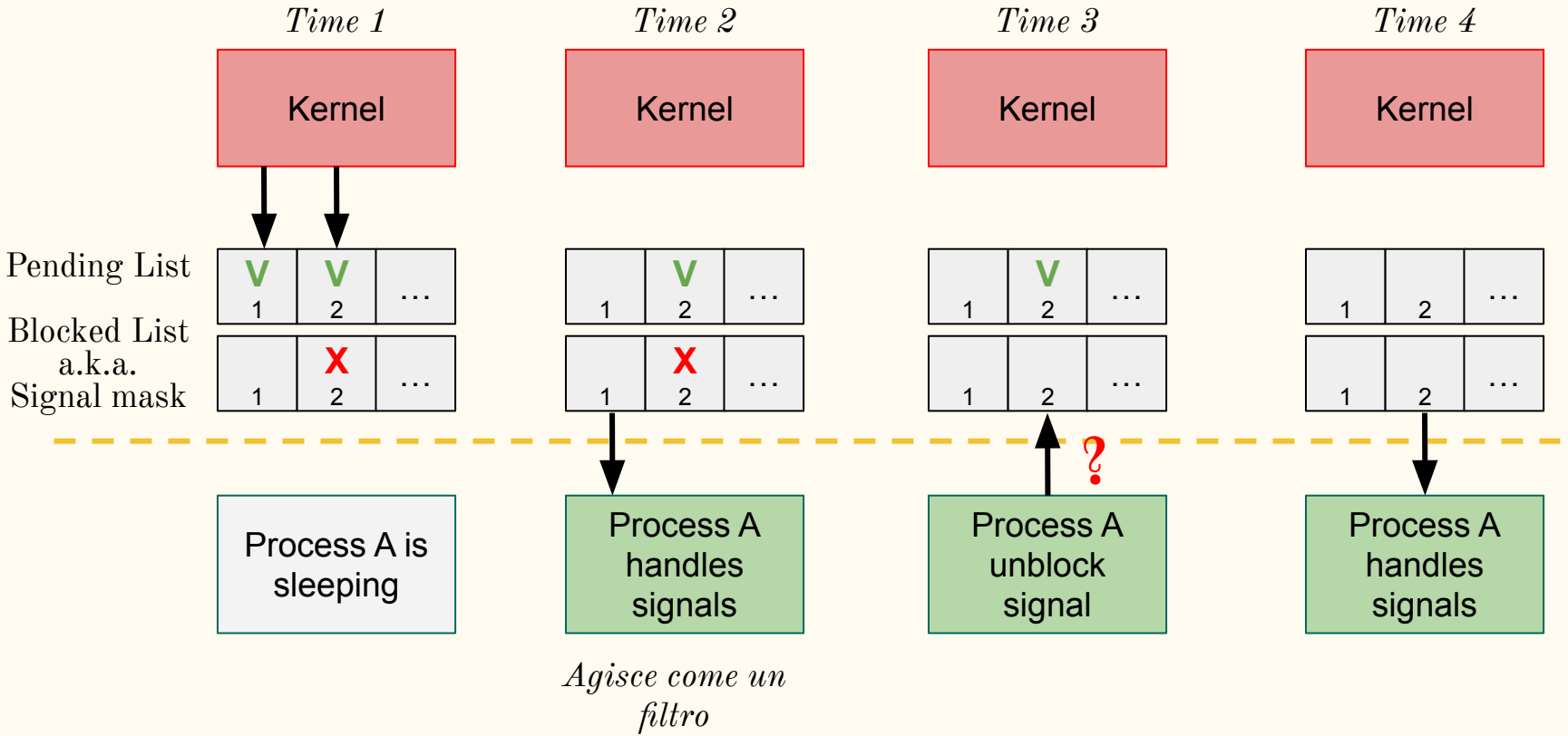
Oltre alla lista dei “pending signal” esiste la lista dei “blocked signals”, ovvero dei segnali ricevuti dal processo ma volutamente non gestiti. Mentre i segnali ignorati non saranno mai gestiti, i segnali bloccati sono solo *temporaneamente* non gestiti.

Al contrario dei segnali ignorati, un segnale bloccato rimane nello stato *pending* fino a quando esso non viene gestito oppure il suo handler tramutato in *ignore*.

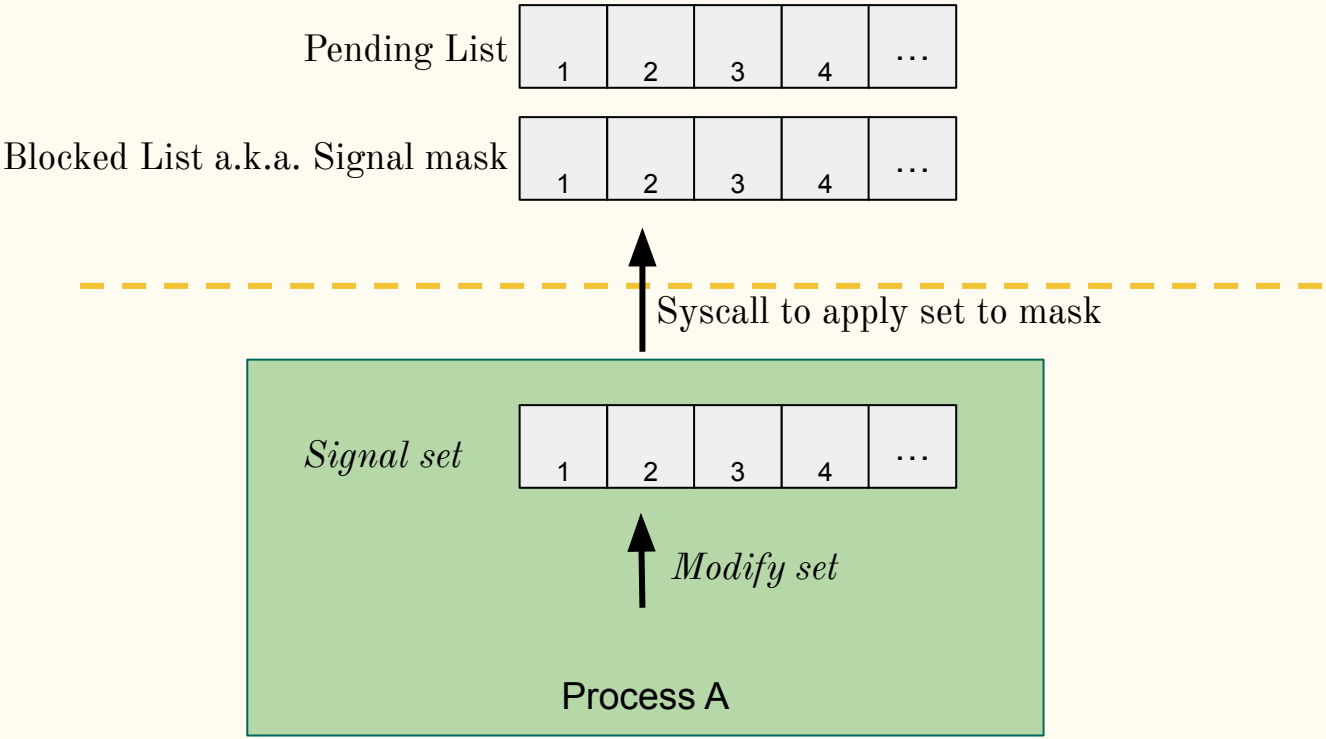
Per allertare il kernel dei segnali che devono essere bloccati, un processo può modificare la propria **signal mask**, ovvero una struttura dati mantenuta nel kernel che può essere alterata con la funzione `sigprocmask()`.

NB: `SIGKILL` e `SIGSTOP` non possono essere bloccati.

Visione concettuale



La signal mask



Bloccare i segnali: sigset_t

La signal mask viene memorizzata come struttura dati ottimizzata che non può essere modificata direttamente. Invece, può essere gestita attraverso un `sigset_t`, cioè una struttura dati locale contenente un elenco di segnali. Questo insieme può essere modificato con funzioni dedicate e poi può essere utilizzato per modificare la maschera di segnale stessa.

NB: la modifica di questa struttura non modifica implicitamente la maschera dei segnali! Le modifiche devono essere salvate con `sigprocmask()`.

Bloccare i segnali: preparare il set

```
int sigemptyset(sigset_t *set);
```
Svuota il set (inizializza)
```
int sigfillset(sigset_t *set);
```
Riempie il set con tutti i segnali (inizializza)
```
int sigaddset(sigset_t *set, int signo);
```
Aggiunge singolo segnale al set
```
int sigdelset(sigset_t *set, int signo);
```
Rimuove singolo segnale dal set
```
int sigismember(const sigset_t *set, int signo);
```
Restituisce 0 se il segnale non è impostato nel set, 1 altrimenti

Bloccare i segnali: applicare il set

```
int sigprocmask(int how, const sigset_t *restrict set,
                sigset_t *restrict oldset);
```

A seconda del valore di `how` e di `set`, la maschera dei segnali del processo viene cambiata. Nello specifico `how =` :

- `SIG_BLOCK`: i segnali in `set` sono aggiunti alla maschera del processo;
- `SIG_UNBLOCK`: i segnali in `set` sono rimossi dalla maschera del processo;
- `SIG_SETMASK`: `set` diventa la maschera del processo.

Se `oldset` non è nullo, in esso verrà salvata la vecchia maschera (anche se `set` è nullo). I segnali in attesa verranno ricevuti **prima** che sigprocmask ritorni**!**

Esempio

```
#include <signal.h>

int main(){
    sigset_t mod,old;
    sigfillset(&mod); // Add all signals to the blocked list
    sigemptyset(&mod); // Remove all signals from blocked list
    sigaddset(&mod,SIGALRM); // Add SIGALRM to blocked list
    sigismember(&mod,SIGALRM); // is SIGALRM in blocked list?
    sigdelset(&mod,SIGALRM); // Remove SIGALRM from blocked list

    // Update the current mask with the signals in ‘mod’
    sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Esempio 2

```
$ kill -10 <PID> # ok
$ kill -10 <PID> # blocked
```

```c
#include <signal.h> <unistd.h> <stdio.h>          //sigprocmask.c
sigset_t mod, old;
int i = 0;
void myHandler(int signo){
    printf("signal received\n");
    i++;
}
int main(){
    printf("my id = %d\n",getpid());
    signal(SIGUSR1,myHandler);
    sigemptyset(&mod); //Initialise set
    sigaddset(&mod,SIGUSR1);
    while(1) if(i==1) sigprocmask(SIG_BLOCK,&mod,&old);
}
```

Verificare pending signals: sigpending()

```
int sigpending(sigset_t *set);
```

```
//sigpending.c
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

sigset_t mod,pen;
void handler(int signo){
    printf("SIGUSR1 received\n");
    sigpending(&pen);
    if(!sigismember(&pen,SIGUSR1))
      printf("SIGUSR1 not pending\n");
    exit(0);
}
```

```
int main(){
    signal(SIGUSR1,handler);
    sigemptyset(&mod);
    sigaddset(&mod,SIGUSR1);
    sigprocmask(SIG_BLOCK,&mod,NULL);
    kill(getpid(),SIGUSR1);
    // sent but it's blocked...
    sigpending(&pen);
    if(sigismember(&pen,SIGUSR1))
        printf("SIGUSR1 pending\n");
    sigprocmask(SIG_UNBLOCK,&mod,NULL);
    while(1);
}
```

sigaction() da usare al posto di signal()

```
int sigaction(int signum, const struct sigaction *restrict
              act, struct sigaction *restrict oldact);

struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; //Signals blocked during handler
    int sa_flags; //modify behaviour of signal
    void (*sa_restorer)(void); //Deprecated, not POSIX
};
```

NB: i flags della variabile sa_flags dovrebbero essere sempre inizializzati almeno a 0!

Esempio

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>   //sigaction.c
void handler(int signo){
    printf("signal received\n");
}
int main(){
    struct sigaction sa; //Define sigaction struct
    sa.sa_handler = handler; //Assign handler to struct field
    sa.sa_flags = 0; //Initialise flags
    sigemptyset(&sa.sa_mask); //Define an empty mask
    sigaction(SIGUSR1,&sa,NULL);
    kill(getpid(),SIGUSR1);
}
```

Esempio: bloccare segnali

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction2.c
void handler(int signo){
    printf("signal %d received\n",signo);
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sa.sa_flags = 0; //Initialise flags
    sigemptyset(&sa.sa_mask); //Use an empty mask → block no signal
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: bloccare segnali

```
$ kill -10 <PID> ; sleep 1
  && kill -12 <PID>
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h> //sigaction3.c
void handler(int signo){
    printf("signal %d received\n",signo);
    sleep(2);
    printf("Signal done\n");
}
int main(){
    printf("Process id: %d\n",getpid());
    struct sigaction sa;
    sa.sa_handler = handler;
    sa.sa_flags = 0; //Initialise flags
    sigemptyset(&sa.sa_mask);
    sigaddset(&sa.sa_mask,SIGUSR2); // Block SIGUSR2 in handler
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio: sa_sigaction

```
$ echo $$ ; kill -10 <PID> # custom
$ kill -10 <PID> # default
```

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigaction4.c
void handler(int signo, siginfo_t * info, void * empty){
    //print id of process issuing the signal
    printf("Signal received from %d\n",info->si_pid);
}
int main(){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO; // Use sa_sigaction
    sa.sa_flags |= SA_RESETHAND; // Restore def handler afterward
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Inviare un payload con un segnale

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Invia un segnale `sig` al processo identificato da `pid`, accompagnato da un payload `value`. Quest'ultimo è rappresentato con la seguente union:

```
union sigval {
    int   sival_int;
    void *sival_ptr;
};
```

Essa può contenere un puntatore qualsiasi o un intero, che verrà ricevuto solo ed esclusivamente se il processo destinatario utilizza `sigaction` con `SA_SIGINFO`.
NB: il processo ricevente non condivide la memoria, quindi non può essere usato per scambiare dati tra processi diversi.

Esempio ricezione

```c
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigqueueR.c
void handler(int signo, siginfo_t * info, void * empty){
    printf("Received integer\n",info->si_value.sival_int);
    exit(0);
}
int main(){
    struct sigaction sa;
    sa.sa_sigaction = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_SIGINFO; // Use sa_sigaction
    sigaction(SIGUSR1,&sa,NULL);
    while(1);
}
```

Esempio invio

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>  //sigqueueS.c

int main(int argc, char ** argv){
    union sigval value;
    value.sival_int = atoi(argv[2]);
    sigqueue(atoi(argv[1]),SIGUSR1,value);
    while(1);
}
```

E se provassimo ad inviare una struct?

```
#include <signal.h> <unistd.h> <stdlib.h> <stdio.h>
typedef struct{
    char name[45];
    int age;
} Person;

int main(int argc, char * argv[]){
    Person p;
    strncpy(p.name,"Michele",44);
    p.age = 28;
    union sigval value;
    value.sival_ptr = &p;
    sigqueue(atoi(argv[1]),SIGUSR1,value);
}
```

Mettere in pausa: pause() o sigsuspend()

```
int pause();

int sigsuspend(const sigset_t *mask);
```

Mette il processo in pausa fino alla ricezione di un segnale. Sigsuspend consente di impostare una maschera dei segnali temporanea, che verrà sostituita con quella precedente appena la funziona ritorna.

Utili per i busy wait → invece di mettere un while(1), mettiamo:

```
while(1) pause();

while(1) sigsuspend(NULL)
```

Questo consente di conservare i cicli di CPU!

Mettere in pausa: pause()

```c
#include <signal.h> <unistd.h> <stdio.h>                    //pause.c

void myHandler(int sigNum){
    printf("Continue!\n");
}
int main(){
    signal(SIGCONT,myHandler);
    signal(SIGUSR1,myHandler);
    while(1){
        pause();
    }
}
```

```
$ gcc pause.c -o pause.out
$ ./pause.out
# On new window/terminal
$ kill -18/-10 <PID>
```

Mettere in pausa: sigsuspend()

```c
#include <signal.h> <unistd.h> <stdio.h>          //sigsuspend.c

void myHandler(int sigNum){
    printf("Continue!\n");
}
int main(){
    signal(SIGCONT,myHandler);
    signal(SIGUSR1,myHandler);
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask,SIGCONT);
    while(1) sigsuspend(&mask);
}
```

```
$ gcc sigsuspend.c -o sigsuspend.out
$ ./sigsuspend.out
# On new window/terminal
$ kill -18/-10 <PID>
```

Real-time signals

Oltre agli standard signals, esistono anche i real-time signals `SIGRTMIN`, `SIGRTMIN+1`, `SIGRTMIN+2`, …, `SIGRTMAX`.

	Standard	**Real-time**
Nome	statico	Con offset dinamico
Coda segnali uguali	no → solo il primo segnale viene gestito	Separata per ogni segnale
Ordine di gestione	non-specificato	Stesso segnale -> FIFO Diverso segnale -> numero + alto
Re-entrant	sempre	Stesso segnale non può interrompere l'handler

```c
#include <signal.h> <unistd.h> <stdio.h> <stdlib.h>
void handler(int sig, siginfo_t *si, void *ucontext) {
  printf("Handling signal %d with value %d\n", sig, si->si_value.sival_int);
  sleep(2);  // Simulate long handler
}
int main(int argc, char *argv[]) {
  sigset_t mod;
  struct sigaction sa;
  sa.sa_sigaction = handler;
  sa.sa_flags = SA_SIGINFO; // Use sa_sigaction
  sigemptyset(&sa.sa_mask);
  sigaction(SIGRTMIN,&sa,NULL);
  sigemptyset(&mod); sigaddset(&mod,SIGRTMIN);
  sigprocmask(SIG_BLOCK,&mod,NULL); // Block SIGRTMIN
  sigqueue(getpid(),SIGRTMIN,(union sigval){.sival_int = 10});
  sigqueue(getpid(),SIGRTMIN,(union sigval){.sival_int = 20});
  sigprocmask(SIG_UNBLOCK,&mod,NULL); //Unblock SIGRTMIN
  while(1) pause();
}
```

Ed infine... vi ricordate lo status dei figli?

```
void sigHandler(int signo){ exit(59);}   //waitStatus.c
int main() {
    int fid=fork(), wid, st; // Generate child
    if (fid==0) {  //If it is child
        signal(SIGUSR2,sigHandler);
        while(1) pause();
    } else { // If it is parent
        kill(fid,SIGUSR1); //LET'S CHANGE THIS TO SIGUSR2
        wid=waitpid(fid,&st,0);  // wait for ONE child to terminate
        printf("EXITSTATUS: %d\nIFCONTINUED: %d\nIFEXITED: %d\nIFSIGNALED:
%d\nIFSTOPPED: %d\nSTOPSIG: %d\nTERMSIG:%d\n",
        WEXITSTATUS(st),WIFCONTINUED(st),WIFEXITED(st),WIFSIGNALED(st),
        WIFSTOPPED(st),WSTOPSIG(st),WTERMSIG(st));
    }
}
```