Signals POSIX

Montse Farreras i Jordi Fornés

8/11/2021

Índex

1 Introducció **1**

2 Enviant signals **2**

3 Revent (i tractant) signals **3**

4 Bloquejant signals **4**

5 Més sobre terminologia **6**

6 Exemples **7**
6.1 Enviant un signal 7
6.2 Esperant un signal 7
6.3 Esperant dos signals 9
6.4 Herència 10

7 Comentaris finals **10**

8 Resum de funcions **12**

1 Introducció

Els *signals* informen els processos de l'ocurrència d'esdeveniments asíncrons. Estan a UNIX des de les primeres versions, però han sofert canvis importants fins a arribar a l'estandardització de POSIX.

Originalment, tenien l'objectiu de tractar esdeveniments excepcionals, tals com intents d'un usuari de matar el procés d'un altre. En cap cas la idea era dissenyar un mecanisme d'intercomunicació entre processos [McKusick (2005)]. En conseqüència, els primers signals eran fàcils de perdre i difícils de gestionar. Tant BSD (Berkeley Software Distributions) com SVR3 (System V Release 3, d'AT&T) introduiren canvis en el model per aconseguir signals *reliable*, però ambues aproximacions eren incompatibles. El model de POSIX.1[1] que expliquem aquí solucionà el problema. Tot i això, encara perdurem esquemes històrics que poden funcionar per situacions concretes; en general no és una bona idea barrejar els models.

[1] També conegut com IEEE Std 1003.1-1988.

2 Enviant signals

Un **signal** és una interrupció software. És un esdeveniment que es produeix de forma asíncrona i que serveix per a notificar a un procés una situació especial que s'hagi produït durant l'execució d'aquest procés, o bé d'un altre procés. Com que son events asíncrons, interrompen immediatament el procés que s'està executant en el precís moment que succeeix.

Un signal pot ser **enviat** a un procés de tres maneres diferents:

1. Fent servir la *syscall* `int kill(pid_t pid, int sig);` on podem enviar qualsevol dels signals existents en el nostre SO, a qualsevol procés que s'estigui executant en aquell moment[2]. Veure exemple del llistat 1.

2. Des del terminal. Amb *shortcuts*, per exemple pulsant Ctrl+C, matem el procés[3]. O bé amb la línia de comandes `kill -<signum> <pid>` . Per exemple, amb `kill -9 1223` matem el procés 1234.

3. Automàticament: tot procés pare reb un signal quan el seu fill ix. També pot rebre's a través d'un temporitzador que esgota el seu temps i és llançat, o a través d'una excepció hardware, on el nostre kernel també llença un signal, per tal d'avisar al procés que s'hi ha produït aquella excepció.

> **Definició:** Un signal és una interrupció software que el kernel envia a un procés per a informar-lo d'algun esdevenimentt asíncron o d'alguna situació especial. Un signal és un event asíncron que ofereix una determinada informació.
>
> - asíncron: un procés pot rebre el signal en qualsevol moment durant la seva execució, en qualsevol part del codi. Un event síncron seria aquell que passa cada cop que s'executa una determinada instrucció.
> - informació: rebuda als arguments de la rutina d'atenció al signal.

El nombre de signals varia entre sistemes[4], però els 19 del UNIX System V (R2) poden ser classificats segons el seu origen:

- Finalització d'un procés, enviat quan un procés *exits*.
- Excepcions, enviades quan es produeix una anomalia tal com un procés accedint a una adreça fora del seu espai, intentant escriure a una posició de memòria de només lectura o produïdes per instruccions privilegiades o errors de hardware.
- Situacions no recuperables durant una *syscall*, com exhaurir els recursos del sistema durant un `exec`.

[2] En Linux, per veure la taula de signals executeu al terminal, `kill -L` i per traduir del número de signal al nom del signal feu servir `kill -l <signum>`.

[3] Podeu consultar els més comuns picant al terminal `stty -a | grep -Ewoe '(intr|quit|susp) = [^ ;]+'`.

[4] Entre unixes i arquitectures, però POSIX.1-2018 defineix 29 signals que han d'estar implementats. Fent `kill -l` a Linux trobem 64 signals definits i fent el mateix a macOS apareixen 31.

- Errors inesperats durant una *syscall*, com escriure a una `pipe` sense lectors, o fer servir una referència ilegal a un `lseek`.
- Per un procés en mode usuari, quan fa servir la funció `alarm` o la *syscall* `kill`.
- Pel terminal, degut a la interacció amb l'usuari; com quan aquest pica segons quines tecles.
- Per rastrejar (*tracing*) l'execució d'un procés.

En resum, els signals poden ser provocats pel kernel, per un altre procés o pel propi procés.

El procés pot testejar la variable `errno` per a veure si s'ha produït un signal mentre s'estava executant una sentència de codi que ha retornat error.

3 Revent (i tractant) signals

El kernel tracta els signals dins el context d'execució del procés l'ha rebut (ergo, el procés s'ha d'estar executant per a rebre un signal). Hi ha tres possibles tractaments: actuació per defecte (`SIG_DFL`)[5], el procés ignora el signal (`SIG_IGN`) o el procés executa una funció d'usuari, previament associada al signal via la *syscall* `sigaction()`.

La crida al sistema `sigaction()` té aquesta signatura (feu `man sigaction`):

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

Quan un programa comença la seva execució l'estat dels seus signals és `SIG_DFL` o bé `SIG_IGN`.

La *syscall* `exec` canvia el disposició de tots els signals que estaven capturats i la passa a `SIG_DFL`[6]. En canvi, quan un procés comença via `fork()`, hereta les disposicions dels signals del seu pare. Noteu que en aquest cas, el fill té una còpia del codi del pare que inclou, esclar, les funciones de tractament als signals capturats via `sigaction()`.

Aquest `struct sigaction`, en algunes arquitectures es defineix com una `union`:

```
struct  sigaction {
     union __sigaction_u __sigaction_u;  /* signal handler */
     sigset_t sa_mask;               /* signal mask to apply */
     int     sa_flags;               /* see signal options below */
};
union __sigaction_u {
     void    (*__sa_handler)(int);
     void    (*__sa_sigaction)(int, siginfo_t *,
                               void *);
};
```

[5] Habitualment el comportament per defecte és que el procés cridi a `exit()` en mode kernel.

[6] Obvi, donat que `exec` canvia el codi del procés i ja no existeixen les funcions assignades via `sigaction()`

Però en d'altres no:

```
 struct sigaction {
     void     (*sa_handler)(int);
     void     (*sa_sigaction)(int, siginfo_t *, void *);
     sigset_t   sa_mask;
     int        sa_flags;
     void     (*sa_restorer)(void);
};
```

En qualsevol cas, només podeu omplir un dels dos camps: `sa_handler` o bé `sa_sigaction`[7]. Tant un com l'altre tenen un argument de tipus `int` que identifica el signal rebut.

La funció `sigaction(signum,*act, *oldact)` permet consultar i/o modificar el tractament associat a un signal en concret. Veiem els seus arguments un per un:

- `signum` és el signal del qual volem consultar/modificar una acció (la rutina d'atenció al signal o RAS),
- `*act` és un punter, si no és *null* contindrà la nova RAS.
- Si `*oldact` no és `NULL`, en tornar de la funció, contindrà la RAS prèvia a la crida. A diferència d'implementacions històriques de signals, POSIX requereix que, un cop tornats de la crida a `sigaction()`, la rutina d'atenció al signal romangui la mateixa fins que es torni a cridar a `sigaction()`.

Ara examinem més detingudament l'`struct sigaction`. Omplirem, tres camps. El primer és `sa_handler` apunta a la rutina d'atenció al signal. El segon, `sa_mask`, és una màscara de bits, un per signal. Aquesta màscara s'afegirà a la màscara de signals bloquejats del procés (la *signal mask*), només mentre s'estigui executant la rutina apuntada per `sa_handler`. D'aquesta manera podem bloquejar certs signals només mentre s'estigui executant la rutina d'atenció al signal. El tercer i últim argument, `sa_flags` especifica una sèrie d'opcions en el comportament del tractament del signal. Llistem només algunes (feu `man sigaction` per veure-les totes.).

Opció	Descripció
SA_RESETHAND	Restaura el tractament per defecte després de la ras.
SA_RESTART	Reinici de funcions bloquejants després de la ras.
SA_SIGINFO	La rutina d'atenció al signal té tres arguments.

4 Bloquejant signals

El procés pot controlar quan i en quin ordre li arriben els signals mitjançant tres *syscalls*[8]: `sigprocmask, sigsuspend` i `sigaction`.

[7] En aquest document farem servir només `sa_handler`, però noteu que `sa_sigaction` és molt més potent, car permet al procés que reb el signal obtenir molta més informació.

[8] Recordeu, les *syscalls* estan al capítol 2 del `man`. En canvi, les funcions de llibreria, tals com `sigemptyset, sigfillset, sigaddset, sigdelset` o `sigismember` estan al capítol 3. Això si, totes les funcions que comencen per `sig` estan a `signal.h`.

sigprocmask → La funció `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);` permet consultar i/o modificar la màscara de signals bloquejats del procés. Analitzem els seus arguments:

`how` Indica com s'aplica el segon argument. Pot ser `SIG_BLOCK`, els signals de `set` s'afegeixen a la *signal mask*, és a dir, `signal_mask = signal_mask | *set`. Pot ser `SIG_UNBLOCK`, els signals de `set` s'esborren de la *signal mask*, és a dir, `signal_mask = signal_mask & ~ (*set)`. Finalment, també pot ser `SIG_SETMASK`, en aquest cas, `signal_mask = *set`.

`set` és l'adreça d'una màscara de bits. Si és `NULL` no modificaren la *signal mask*, només la consultarem, amb el tercer argument.

`oldset` és l'adreça d'una màscara de bits. Si no és `NULL`, en tornar de la funció, apuntarà a la *signal mask* que tenia el procés abans de crida a la funció.

Per manipular una màscara de bits (que eventualment podrà esdevenir la *signal mask*, en cridar a `sigprocmask`), teniu una sèrie de funciones de llibreria. Feu `man` de qualsevol de les següents:

```
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);
```

Per veure com es fan servir aquestes funcions, feu una ullada al llistat 4. A la línia 13 es declaren les màscares de bits anomenades `mask` i `oldmask`. A la línia 16, s'esborra qualsevol signal bloquejat de `mask`. A la línia 17, es bloqueja el signal `SIGUSR1` de `mask`. Finalment, a la línia 18 es modifica la *signal mask* en cridar `sigprocmask` passant, com a segon argument, l'adreça de `mask`.

sigsuspend → De vegades volem esperar un signal en concret. Tradicionalment, això es feia desbloquejant el signal que volem esperar i cridant a `pause()`, però si els signal arriba just entre les dues accions, aquest es perd. Un cas semblant el tenim a l'exemple del llistat 3, a on el signal pot arribar entre les línies 12 i 13 (recordeu que cadascuna pot correspondre a més d'una línia d'assemblador). En aquest exemple, si la condicio és certa (perquè encara no havia arribat el signal), i aquest arriba just després d'avaluar la condició, estarem bloquejats al `pause()` per sempre (o fins que arribi un altre signal, però el que voliem era detectar el primer signal). La manera segura és que el desbloquejat i la pausa es produeixin en una única acció. Per això es va introduïr `int sigsuspend(const sigset_t *mask);`

El que fa `sigsuspend()` és intercanviar, de manera atòmica, la *signal mask* del procés per `mask`. El procés quedarà suspès fins que arribi un

signal dels capturats (no bloquejats a `mask`) o algun dels que provoquen terminar el procés. Si el procés torna de la RAS, llavors `sigsuspend()` acaba tornant a deixar la *signal mask* tal com estava i retornant -1 (i posant `errno` a `EINTR`) i el procés continua amb la següent instrucció.

A la secció 6.2 i següent, teniu dos exemples d'ús de `sigsuspend()`. Noteu que aquesta *syscall* sempre retorna -1 (si retorna del handler) i només canvia la *signal mask* mentre el procés està bloquejat. Trobareu més exemples al llibre [Stevens (2013)].

sigaction → Per últim, ja hem vist que a l'estructura de dades del `sigaction` tenim un camp anomenat `sa_mask`, que és una màscara de bits, un per signal. Aquesta màscara s'afegirà a la màscara del procés (*signal mask*), només mentre s'estigui executant la rutina apuntada per `sa_handler`.

5 Més sobre terminologia

Aquí teniu un intent de traducció al català de diferents termes del camp semàntic dels signals:

signal Senyal que el SO envia al procés en execució per tal de notificar-li que s'ha produït un cert event.

generate/send El signal s'envia al procés que se li indiqui al SO quan s'ha produït un cert event.

deliver S'utilitza per indicar que un signal s'ha dipositat. És a dir, que el signal ha estat atès pel procés que l'ha rebut. S'ha executat la rutina d'atenció al signal.

lifetime El temps de vida d'un signal és el temps que passa entre que el signal s'ha generat (per un cert event) fins que aquest s'ha dipositat (*delivered*).

pending Diem que un signal està pendent, quan s'ha generat, però no ha sigut atès pel procés receptor.

catch Un signal està atrapat o capturat quan ha estat reconfigurada la seva RAS i per tant s'executa aquesta (la definida pel mateix usuari) en comptes de la per defecte.

handle És el que ens indica quina rutina s'ha d'executar quan és dipositat un determinat signal.

signal handler Rutina d'atenció al signal (RAS). Rutina on l'usuari definirà les accions que s'han de produir si es diposita un determinat signal.

ignore Un signal pot ser ignorat. És a dir, pot generar-se i ser dipositat, però serà com si no s'hagués produït. No s'atendrà ni a la seva actuació per defecte ni a l'actuació que hagués pogut definir l'usuari a la RAS.

blocked Un signal pot ser bloquejat. Això ens permetrà protegir zones del codi on rebre un signal podria provocar un mal funcionament del programa. Mentre un signal estigui bloquejat no serà dipositat, podent ser dipositat només quan es desbloquegi.

set of blocked signals Conjunt de senyals bloquejats en un instant determinat.

suspend Un procés pot estar en estat de suspend (congelat, bloquejat), en un determinat punt del codi, a l'espera d'uns determinats signals. És molt útil en el cas de un procés que esperi un cert temporitzador o time-out.

6 Exemples

Recordeu: Un signal és una interrupció software que un procés o el sistema operatiu envia a un procés per a informar-lo d'algun event asíncron o d'alguna situació especial.

6.1 Enviant un signal

Ja hem vist que la *syscall* per enviar una signal és `int kill(pid_t pid, int sig)`, consulteu el manual capítol 2 per esbrinar, per exemple, què passa quan els arguments són zero. El llistat 1 és un exemple molt simple d'ús d'aquesta crida.

Llistat 1: Envia USR1 al procés que li passem com argument.

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

int
main(int argc, char **argv)
{
  if (argc < 2) {
    write(1, "pid!\n", 5);
    exit(1);
  }
  if (kill(atoi(argv[1]), SIGUSR1) < 0) {
    perror("No puc enviat USR1 al pid que m'has passat");
  }
}
```

6.2 Esperant un signal

La forma més simple és fer servir una espera activa. Tal com mostra el llistat 2. Podem substituir el `for(;;);` per un `pause()` i deixarem de tenir una espera activa. El problema bé quan tractament les mateixes variables tant a la rutina d'atenció al signal com al programa principal. Imagineu que volem esperar que arribi el signal USR1 i fem servir la variable `usr_interrupt` que s'activarà a la rutina d'atenció al signal.

Però si el signal arriba després de consultar-la però abans del `pause()` i no arriba cap altre signal, esperarem per sempre.

La forma correcta d'esperar un signal, tot i que ens obliga a escriure més codi, és amb `sigsuspend`. La crida a sistema `sigsuspend` permet bloquejar al procés que la crida a l'espera d'un subconjunt de signals. De manera atòmica bloqueja el conjunt de signals passats com a paràmetre i espera el dipòsit d'algun signal

Llistat 2: Espera activa per l'arribada del signal USR1.

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

void
ras(int s)
{
  write(1, "Rebut!\n", 7);
}
int
main()
{
  sigset_t  mask  , oldmask;
  struct sigaction sa;
  /* Bloquejant el signal fins que ho tinguem tot preparat */
  sigemptyset(&mask);
  sigaddset(&mask, SIGUSR1);
  sigprocmask(SIG_BLOCK, &mask, &oldmask);
  sa.sa_handler = ras;
  sigaction(SIGUSR1, &sa, NULL);
    /* esperant que arribi el signal */ //
    sigprocmask(SIG_SETMASK, &oldmask, NULL);
  for (;;);
}
```

Llistat 3: El signal pot arribar després de consultar la variable.

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
int usr_interrupt=0;
void caught(int s) {
  usr_interrupt=1;
}
int main() {
struct sigaction sa;
sa.sa_handler=caught;
sigaction(SIGUSR1,&sa,NULL);
if (!usr_interrupt)
  pause ();
}
```

no bloquejat. El paràmetre serà la nova màscara del procés mentre estiguem bloquejats en aquesta crida. Inclourem en el set, tots els signals menys els que volem que ens arribin i per tant facin que ens desbloquegem i seguim executant el nostre programa. Normalment, és utilitzat com a un punt de control, així que el `sigsuspend` estarà dins d'una regió crítica, fent els bloquejos necessaris i restaurant la màscara després. Sinó podria passar que m'arribés el signal que jo espero justament abans d'esperar-lo amb sigsuspend, amb el qual em quedaria bloquejat indefinidament. El valor de retorn és -1 sempre amb `errno = EINTR`. Si el tractament del signal és `SIG_IGN`, aquest signal no ens desbloquejarà d'un `sigsuspend` ni ens provocarà l'error `EINTR` en les crides.

Exemple genèric:

Noteu que el `sigsuspend()` està dins un bucle. Aquest esquema general és per poder esperar exactament a USR1, encara que a la màscara hi hagin més signals desbloquejats.

Llistat 4: Esperant a SIGUSR1.

```c
#include <signal.h>
#include <stdio.h>
int   usr_interrupt = 0;
void
caught(int s)
{
  usr_interrupt = 1;
}
int
main()
{
  sigset_t  mask  , oldmask;
  struct sigaction sa;
  /* Bloquejant el signal fins que ho tinguem tot preparat */
  sigemptyset(&mask);
  sigaddset(&mask, SIGUSR1);
  sigprocmask(SIG_BLOCK, &mask, &oldmask);
  sa.sa_handler = caught;
  sigaction(SIGUSR1, &sa, NULL);
  /* esperant que arribi el signal */
  while (!usr_interrupt)
    sigsuspend(&oldmask);
  sigprocmask(SIG_UNBLOCK, &mask, NULL);
  return 0;
}
```

6.3 Esperant dos signals

Un altre exemple, vull esperar que arribi un USR1 o un USR2: llistat 5.

Llistat 5: Esperant a SIGUSR1 o a SIGUSR2.

```c
void caught(int s) {
  char buff[32];
  int usr;
  if (s==SIGUSR1)
    usr = 1;
  else if (s==SIGUSR2)
    usr = 2;
  sprintf(buff,"USR%d\n",usr);
  write(1,buff,strlen(buff));
}
int main() {
sigset_t mask, oldmask, set;
struct sigaction sa;
/* Bloquejant el signal fins que ho tinguem tot preparat */
sigemptyset (&mask);
sigaddset (&mask, SIGUSR1);sigaddset (&mask, SIGUSR2);
sigprocmask (SIG_BLOCK, &mask, &oldmask);
sa.sa_handler=caught;
sigaction(SIGUSR1,&sa,NULL);
sigaction(SIGUSR2,&sa,NULL);
sigfillset(&set);
sigdelset(&set,SIGUSR1); sigdelset(&set,SIGUSR2);
sigsuspend(&set);
}
```

6.4 Herència

La programació dels signals del pare i la `procmask` passa als fills en fer un `fork()` amb la salvetat que el comptador de l'alarm passa a zero. La programació dels signals es perd en fer un `exec` degut a que s'ha canviat el codi. Exemple, llistat 6.

Llistat 6: El pare rebrà un SIGALRM i el fill no.

```
1  int main() {
2    sigset_t mask;
3    sigemptyset(&mask);
4     sigaddset(&mask, SIGALRM);
5     sigprocmask(SIG_BLOCK,&mask, NULL);
6     alarm(5);
7     sigfillset(&mask);
8     sigdelset(&mask, SIGALRM);
9     fork();
10    sigsuspend(&mask);
11    return 0;
12   }
```

7 Comentaris finals

- Implícitament, el signal que ha arribat està bloquejat durant el seu tractament, és a dir, dins de la rutina d'atenció al signal (*handler*) no podem rebre un altre signal del mateix tipus.
- Si un signal es rep vàries vegades i està bloquejat, només s'executa 1 ras.
- POSIX no especifica l'ordre en el que els signals bloquejats són tractats quan es desbloquegen, però nosaltres assumirem que ho fan en ordre d'arribada.
- Si consultem variables globals que es modifiquen tant en les ras com en el `main`, s'ha de vigilar (zona crítica!!). Protecció amb `sigprocmask`.
- Seguiu els 6 consells, *guidelines*, de [Byant & O'Hallaron (2016)] per gestionar els signals de forma segura:

  G0 Les funcions RAS, quan més simples millor.

  G1 Dins la RAS, crideu només funcions a prova de signals (reentrants o no interrupibles).

  G2 Dins la RAS, deseu i restaureu `errno`.

  G3 Protegiu l'accès a variables globals compartides bloquejant els signals.

  G4 Declareu les variables globals amb `volatile`.

  G5 Declareu les variables globals que modificareu a la RAS i al `main()` amb `volatile sig_atomic_t`.

Referències

[Byant & O'Hallaron (2016)] Bryant, Randal E., O'Hallaron David R. 2016. *Computer systems : a programmer's perspective*. Pearson, 2016.

https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004062589706711

[Kernighan (2020)] Kernighan, Brian. 2020. *UNIX. A History and a Memoir*. Kindle Direct Publishing, 2020.

[Back (1986)] Back, Maurice J. 1986. *The Design of the UNIX Operating System*. Prentice Hall, PTR. 1986.

https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991000067889706711

[Sechrest (1986)] Sechrest, Stuart. *An Introductory 4.3BSD Interprocess Communication Tutorial*, Unix Programmer's Supplementary Documents, Vol. 1 (PS1), 4.3 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, Univ. of California, Berkeley, Calif., Apr 1986.

https://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf

[Kernighan (1984)] Kernighan, B. W., & Pike, R. *The UNIX Programming Environment*. Prentice-Hall, Inc., Englewood Cliffs, NY. 1984

https://discovery.upc.edu/permalink/34CSUC_UPC/rdgucl/alma991000110939706711

[Stevens (2013)] Stevens, W. Richard. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

https://discovery.upc.edu/permalink/34CSUC_UPC/11q3oqt/alma991004011309706711

[McKusick (2005)] McKusick, Marshall Kirk. *The design and implementation of the FreeBSD operating system*. Addison-Wesley, 2005. https://discovery.upc.edu/permalink/34CSUC_UPC/8e3cvp/alma991003102659706711

8 Resum de funcions

Functions to manipulate signal sets

```
sigemptyset(sigset_t *set);
sigfillset(sigset_t *set);
sigaddset(sigset_t *set, int signo);
sigdelset(sigset_t *set, int signo);
```

Functions to manipulate current signal mask

```
sigprocmask(SIG_BLOCK, sigset_t *set, sigset_t *old);}
sigprocmask(SIG_UNBLOCK, sigset_t *set, sigset_t *old);}
sigprocmask(SIG_SETMASK, sigset_t *set, sigset_t *old);}
```

`SIG_BLOCK` The new mask is the union of the current mask and the specified set.

`SIG_UNBLOCK` The new mask is the intersection of the current mask and the complement of the specified set.

`SIG_SETMASK` The current mask is replaced by the specified set.

Therefore, to block a signal:

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set,SIGNAL1);
sigprocmask(SIG_BLOCK, &set, NULL);
```

To restore the process mask:

```
sigprocmask(SIG_UNBLOCK,&set,NULL);
sigprocmask(SIG_SETMASK,&old,NULL);
```

To check the process mask:

```
sigprocmask(SIG_SETMASK, NULL, sigset_t *old);
```

The `sigaction()` system call assigns an action for a signal specified by `signal`.

```
sigaction(int signal, struct sigaction *act, struct sigaction *old);
```

You must previously fill the `act` variable:

```
act.sa_flags
act.sa_mask
act.sa_handler
```

In the field `act.sa_mask` you should include all the signals that will be block during the execution of the handler function. Sending signals

```
kill(int pid, ind signal);
alarm(int seconds);
```

Atomically release blocked signals and wait for interrupt

```
sigsuspend(sigset_t *set);
```

Temporarily changes the blocked signal mask to the set to which `set` points.