EPU - Informatique ROB4
Informatique Système
Introduction aux signaux

Sovannara Hak, Jean-Baptiste Mouret
hak@isir.upmc.fr

Université Pierre et Marie Curie
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2014-2015

Plan de ce cours

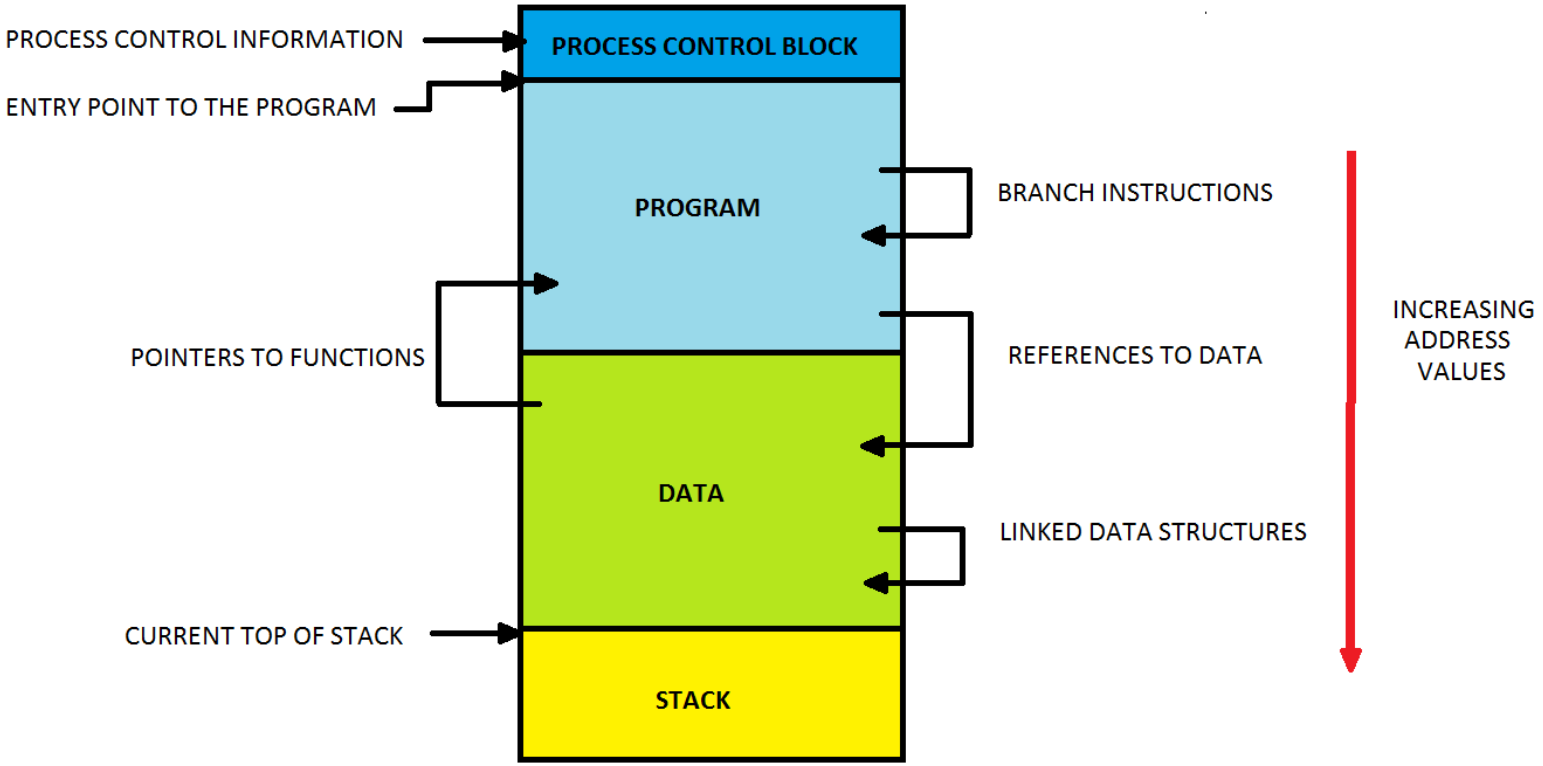
1. Processus : quelques approfondissements
   - Quelques rappels
   - Commutation de contexte et commutation de mode
   - Bloc de contrôle d'un processus (PCB) Linux
2. Signaux
   - Définition
   - Signaux et PCB
   - Envoi et prise en compte d'un signal
   - Prise en compte : quelques détails
3. Programmation des signaux
   - Envoi avec kill ()
   - Blocage de signaux
   - Associer une fonction à un signal
   - Autres fonctions liées aux signaux

Rappel du plan

Processus = programme en cours d'exécution

- Code du programme, ressources, espace d'adressage, ...
- Descripteur de processus : PID, état, processus parent, enfants, fichiers ouverts
- Un ou plusieurs threads
- Peut communiquer via pipes, signaux, réseau, fichiers

↦ **Un processus en mémoire**



/proc - a pseudo-système de fichiers pour les informations de processus

- `/proc/PID` : répertoire pour chaque processus en cours d'exécution de pid PID
- `/proc/PID/maps` : espaces mémoire actuellement assignés et les permissions d'accès
- `/proc/PID/stat` : information de status du processus (utilisé par `ps`)
  - PID, PPID, GID, ..
  - État (R=running, S=sleeping, D=waiting/disk sleep, Z=zombie, T=stopped on a signal, W=paging)
  - Signaux interceptés,
  - Mémoire virtuelle, ...
- `/proc/PID/status` : comme stat mais lisible par un humain
- `/proc/apm` : advanced power management (batterie)
- `/proc/bus` : répertoire des bus installé
- `/proc/devices` : liste des numéros de périphériques majeurs (numéros de pilote)
- `/proc/interrupts` : interruptions par CPU par périphérique IO
- `/proc/kmsg` : kernel messages, utilié par `dmesg`

↦ pour en savoir plus : `man proc`

↦ /proc/25782 (banshee)

```
shak@samaxe:/proc/25782$ cat status
Name: banshee
State: S (sleeping)
Tgid: 25782
Pid: 25782
PPid: 25777
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 256
Groups: 3 4 7 27 111 116 118 1000 1001
VmPeak:  1971352 kB
VmSize:  1905832 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:   100356 kB
VmRSS:   100356 kB
VmData:  1381904 kB
VmStk:       136 kB
VmExe:      2944 kB
VmLib:     33724 kB
VmPTE:      1156 kB
VmSwap:        0 kB
Threads: 26
SigQ: 0/63605
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000000001001000
SigCgt: 00000005a08004ec
```

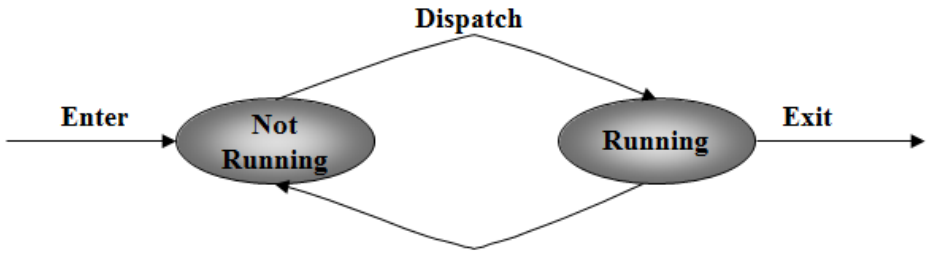
OS - Intéraction des processus

1. L'OS intéragit avec tout les processus en **ordonnançant** les processus et en gérant les cycles de vie des processus (création, exécution, terminaison)
   - maximise l'utilisation CPU pour avoir un temps de réponse raisonnable
   - crucial : affecte la distinction entre hard et soft real time

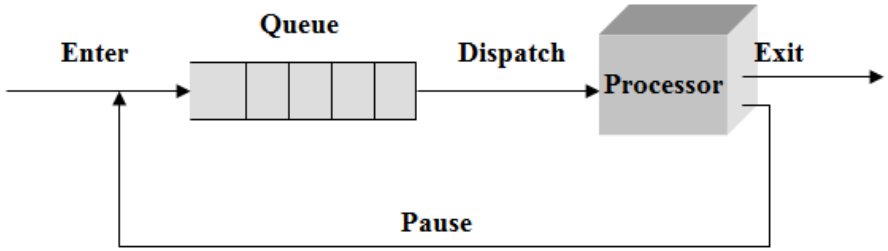
2. Le processus intéragit avec l'OS via des **appels systèmes**, des instructions spéciales mises à disposition par l'OS pour effectuer des opérations "privilegiées".
   - user mode vs kernel mode
   - Appels systèmes peuvent être des lecture/écriture sur des flux, contrôle de processus (kill, wait, stop), etc.

Scheduler vs dispatcher

- Scheduling et dispatching sont effectué habituellement dans la même routine. Ils empêchent un processus de monopoliser le temps processeur.
- **Scheduler (ordonnanceur)** décide quel sera le prochain processus à être exécuté (l'ordre/séquence des processus)
- **Dispatcher** gère la **commutation de processus**.



(a) State transition diagram



(a) Queuing diagram

Rappel du plan

Commutation de contexte et commutation de mode

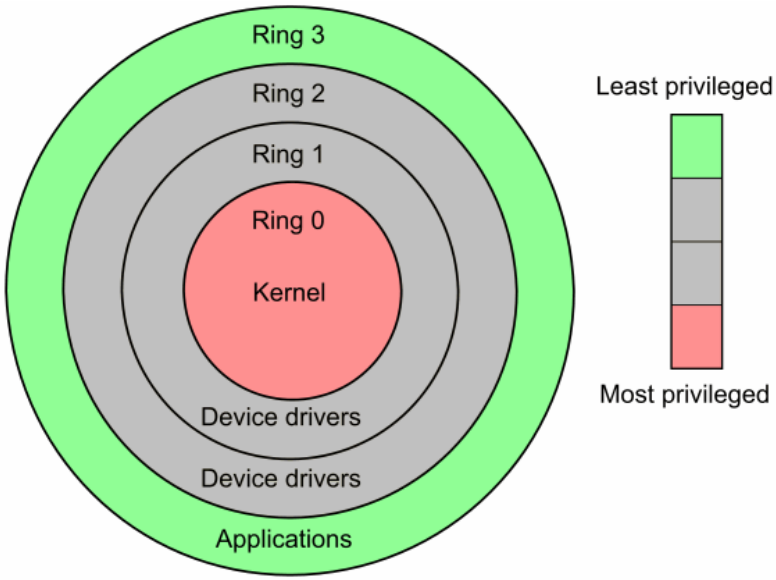
La commutation de contexte est la procédure qui permet de sauvegarder et restaurer l'état d'un processus. Ceci permet d'interrompre et reprendre un processus partager un CPU entre plusieurs processus (avec un certain coût).

Commutation de contexte et commutation/transition de mode

Commutation/transition de mode

Quand une transition entre le mode user et le mode kernel est requise dans l'OS, une commutation de contexte n'est pas nécessaire : une **transition de mode** n'est pas à proprement parler une commutation de contexte.

- Cependant, en fonction de l'OS, une commutation de contexte peu avoir lieu.



Modes

- Kernel mode : privilegié, accès à toute la mémoire et aux ressources système
- User mode : mode initial de tout les programmes

La procédure standard pour effectuer un changement de mode est d'appeler des interruptions logiciels.

Software Interrupts (SWI)

Les SWI sont des mécanisme qui permettent à des processus non privilégié de faire appel à des processus privilegiés :

- elles sont typiquement utilisé pour faire des demandes d'accès IO
- elles sont appelées pour demander au noyau de faire quelque chose pour le processus courant

Un appel à une instruction SWI entraîne :

- une suspension du processus courant
- une **commutation de mode** vers le mode kernel
- une reprise du processus courant

Les SWI sont générées par :

- des instructions spéciales qui déclenchent des interruptions
- exceptions ou trappes (exemple : division par zero)
- des appels systèmes

Hardware Interrupts (HWI)

Les HWI sont utilisées par des périphériques physiques pour faire remonter à l'OS des evénement asynchrones, par exemple :

- données arrivant d'un périphérique externe ou réseau
- contrôleurs disque signalant une lecture/écriture
- appuie touches clavier, utilisation souris
- /proc/interrupts

Les HWI sont associé à des **Interrupt Request (IRQ)**

- referencé par un numéro d'interruption (assigné au matériel, pour que l'OS puisse déterminer quel périphérique a créé l'interruption et quand elle a eu lieu)

IRQ invoque des **Interrupt Service Routines** ou **Interrupt Handlers (ISR)**

- ISR sont des routines exécuté en réponse à une interruption
- Quand un signal IRQ est envoyé par un périphérique vers le processeur, celui-ci finit son instruction courante, puis exécute l'ISR

Rappel du plan

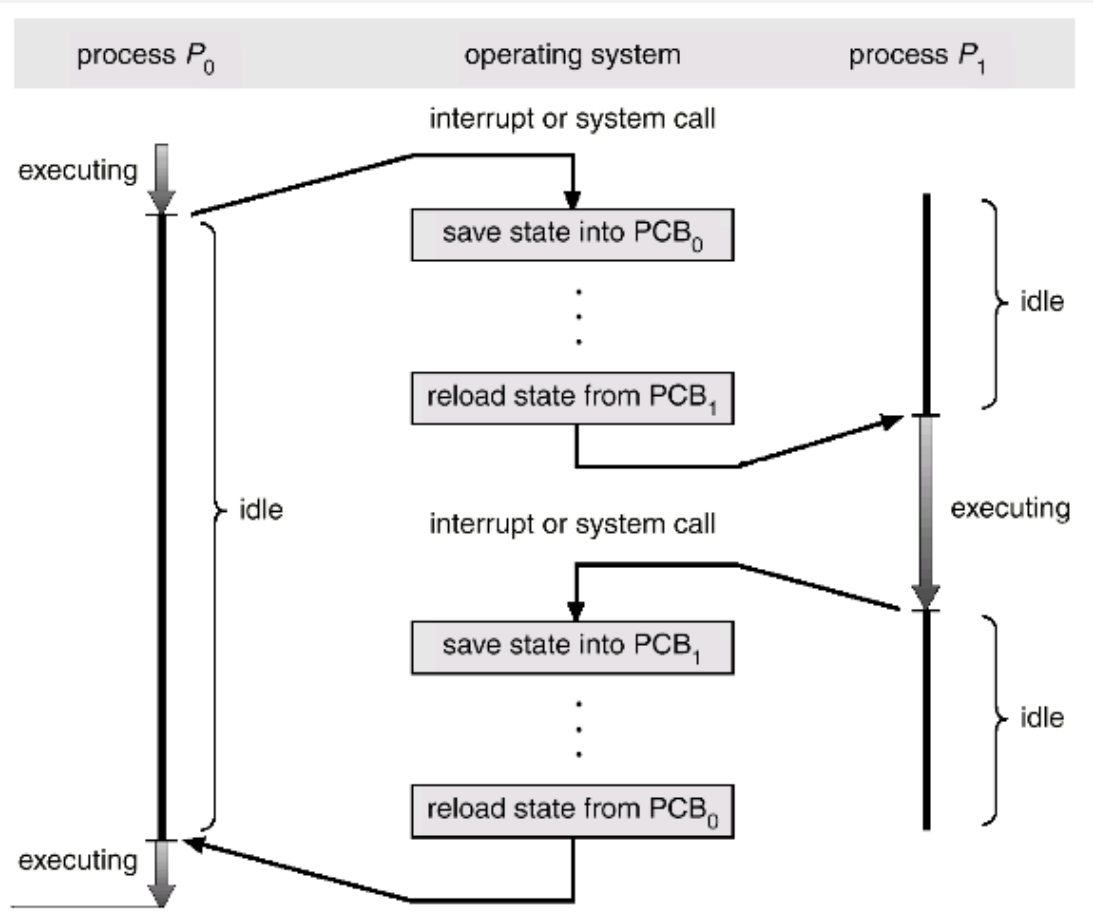
Process Control Block (PCB) or *Task Controlling Struct* or *Task Struct* :

- Chaque processus Linux est représenté par un bloc de contrôle (structure `task_struct` définie dans `linux/sched.h`).
- Ce bloc est alloué dynamiquement par le système au moment de la création du processus.
- Il contient tous les attributs permettant de qualifier et de gérer le processus.
- Chaque bloc de contrôle est accessible depuis une table de processus, chaque entrée de la table étant un pointeur vers un bloc de contrôle.

Informations contenues dans le PCB :

- l'état du processus (prêt/élu, bloqué, zombie, arrêté) et les informations d'ordonnancement (pointer to the next PCB - to be scheduled) ;
- les différents identifiants déjà mentionnés : PID, UID, GID, PPID ainsi que l'identifiant du dernier fils créé et de certains processus frères (cadet le plus jeune et ainé le plus jeune) ;
- les fichiers ouverts par le processus ;
- le terminal attaché au processus ;
- le répertoire courant du processus ;
- le contexte mémoire du processus (address space) ;
- registres, compteur ordinal (adresse de l'instruction en cours) ;
- la partie du contexte processeur pour l'exécution en mode utilisateur ;
- les temps d'exécutions du processus en modes utilisateur et superviseur ainsi que ceux de ses fils ;
- les outils de communication utilisés par le processus parmi lesquels les sémaphores et les **signaux** ;
- les **signaux** reçus par le processus.

Commutation de contexte



Rappel du plan

Définition

- Les signaux sont un mécanisme de communication inter-processus (un message spécial).
- Les signaux sont envoyées à un ou plusieurs processus. Un signal est en général associé à un événement survenu au niveau du système.
- Le message envoyé est un entier qui ne comporte donc pas d'informations propres si ce n'est une correspondance avec l'événement rencontré par le noyau.
- Le processus visé reçoit le signal sous forme d'un flag de son PCB
- La prise en compte du signal par le processus oblige celui-ci à exécuter une fonction de gestion du signal appelé **signal handler**.

- Le noyau Linux admet 64 signaux différents,identifiés par un nombre et décrit par un nom préfixé par la constant SIG.
- Seul le signal 0 ne porte pas de nom (signal NULL). Les signaux 1 à 31 correspondent aux signaux classiques tandis que les signaux 32 à 63 correspondent aux signaux temps-réel. Parmi ces signaux certains ne sont pas définis par la norme POSIX et sont propres à l'OS.

Quelques exemples de signaux

- 2/SIGINT (Interruption du clavier par CTRL C)
- 9/SIGKILL (Terminaison forcé du processus)
- 11/SIGSEGV (Référence mémoire invalide)
- 13/SIGPIPE (Ecriture dans un tube sans lecteur)
- 15/SIGTERM (Terminaison du processus)
- 17/SIGCHLD (Processus fils terminé)
- 18/SIGCONT (Reprise de l'exécution d'un processus stoppé)
- 19/SIGSTOP (Stoppe l'exécution d'un processus)
- 20/SIGSTP (Suspension du processus (CTRL Z)) ...

kill

- La commande `kill` permet d'envoyer des signaux à un processus
- Par défaut cette commande force la terminaison d'un processus
- Le processus à terminer est passé comme argument à la commande `kill` au travers de son PID.

```
shak@samaxe:~$ kill -l
 1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL  5) SIGTRAP
 6) SIGABRT  7) SIGBUS  8) SIGFPE  9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

- man 7 signal

Rappel du plan

Le PCB contient plusieurs champs lui permettant de gérer les signaux :

- Le champ `signal`, de type `sigset_t`, qui stocke les signaux envoyés au processus. Cette structure contient deux entiers sur 32 bits, chaque bit représentant un signal. Une valeur 0 indique que le signal correspondant n'a pas été reçu tandis que la valeur 1 indique que le signal a été reçu.
- le champ `blocked`, de type `sigset_t`, qui stocke les signaux bloqués, i.e. les signaux dont la prise en compte est retardée.
- le champ `sigpending` est un drapeau qui indique s'il existe au moins un signal non bloqué en attente.
- le champ `gsig` qui est un pointeur vers une structure de type `signal_struct` et qui contient notamment pour chaque signal la définition de l'action qui lui est associée.

Rappel du plan

Origine des signaux

- envoi par un autre processus par l'intermédiaire d'un appel à `kill` ;
- envoi par le gestionnaire d'exception qui ayant détecté une trappe à l'exécution du processus positionne un signal pour indiquer l'erreur détectée (par exemple, le gestionnaire d'exception `divide_error()` peut positionner le signal SIGFPE).

Positionnement d'un signal

- Lors de l'envoi d'un signal, le noyau exécute la routine noyau `send_sig_info` ;
- Cette routine positionne à 1 le bit correspondant au signal reçu dans le champ `signal` du PCB du processus destinataire.
- Il existe deux cas particuliers de traitement d'un signal :
  - le signal émis est 0 et la routine retourne immédiatement en indiquant une erreur ;
  - le processus destinataire est zombie et le signal ainsi délivré qualifié de **signal pendant** (pas encore pris en compte).

Prise en compte

- La prise en compte du signal par le processus s'effectue lorsqu'il s'apprête à quitter le mode superviseur noyau pour repasser en mode utilisateur.
- Cette prise en compte est réalisée par la routine noyau `do_signal()` qui traite chacun des signaux pendants du processus.
- Trois types d'actions sont possibles : Ignorer le signal, Exécuter l'action par défaut ou Exécuter une fonction spécifique du programmeur. Les valeurs correspondantes du champ `gsig.sa_handler` du PCB sont SIG_IGN, SIG_DFL et l'adresse de la fonction spécifique à

Rappel du plan

Ignorer le signal

- si le signal est ignoré, aucune action n'est entreprise au moment de sa prise en compte ;
- le signal SIGCHLD (pour reveiller un processus dont un fils vient de mourir) échappe à cette règle

Exécuter l'action par défaut : 5 actions possibles

- term : Abandon du processus (c'est notamment le cas de SIGINT et SIGKILL) ;
- core : Abandon du processus et création d'un fichier **core** (`man 5 core`) contenant son contexte d'exécution exploitable pour le déboggage (c'est notamment le cas de SIGFPE et SIGSEGV) ;
- ign : Signal ignoré (cf. le traitement de SIGCHLD) ;
- stop : Processus stoppé (SIGSTOP, SIGSTP ...) ;
- cont : Processus redémarré (SIGCONT).

Exécution d'une fonction spécifique

Tout processus peut mettre en oeuvre un traitement spécifique pour chacun des signaux hormis le signal 9 (SIGKILL).

Rappel du plan

Envoi d'un signal

- L'envoi d'un signal à un processus se fait avec la fonction kill () définie dans signal .h et dont le prototype est : **int** kill (pid_t pid , **int** num_sig); .
- L'interprétation du résultat diffère en fonction de la valeur pid :
  - pid > 0 : le signal num_sig est envoyé au processus d'identifiant pid ;
  - pid $= 0$: le signal num_sig est envoyé à tous les processus du groupe de processus (champs process group id) auquel appartient le processus appelant ;
  - pid < 0 et pid $\neq$ -1 : le signal num_sig est envoyé à tous les processus du groupe de processus auquel appartient le processus d'identifiant abs(pid) ;
  - pid $= -1$: le signal num_sig est envoyé à tous les processus du système sauf le processus 1 et le processus appelant.
- Les processus doivent avoir la permission d'envoyer un signal (sous Linux : avoir la capacité `CAP_KILL`), ou le processus qui envoie le signal doit avoir le même user ID que le processus cible.
- Trois cas d'échecs sont possibles (perror) :
  - EINVAL : Signal spécifié invalide
  - EPERM : Droits insuffisants
  - ESRCH : PID non existant
- Il existe une version shell de kill .

Rappel du plan

Manipulation du vecteur de signaux

On peut bloquer la réception de signaux en utilisant des vecteurs de signaux du type `sigset_t`, manipulable via les fonctions suivantes :

- `int sigemptyset(sigset_t *ens); /* raz */;`
- `int sigfillset (sigset_t *ens) /* ens = 1,2,..., NSIG */;`
- `int sigaddset(sigset_t *ens, int sig) /* ens = ens + sig */;`
- `int sigdelset (sigset_t *ens, int sig) /* ens = ens − sig */.`

Ces fonctions retournent -1 en cas d'échec et 0 sinon.
`int sigismember(sigset_t *ens, int sig);` retourne vrai si le signal appartient à l'ensemble.

La fonction sigprocmask()

La fonction `int sigprocmask(int op, const sigset_t *nouv, sigset_t *anc);` définie dans `signal.h` permet de manipuler le masque de signaux du processus.
Opération `op` :

- SIG_SETMASK : affectation du nouveau masque `nouv` ;
- SIG_BLOCK : union des deux ensembles `nouv` et `anc` ;
- SIG_UNBLOCK : "soustraction" `anc - nouv`.

Renvoie 0 en cas de succès et -1 sinon.

Rappel du plan

Cette association peut se faire au travers de l'appel à `sigaction()` qui est la version normalisée POSIX de l'appel `signal` dont l'emploi peut s'avérer peu portable.
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);` est défini dans `signal.h`.
Les différents paramètres ont la signification suivante :

- `signum` : numéro de signal pour lequel une fonction spécifique des traitement est spécifiée ;
- `const struct sigaction *act` : définition de la nouvelle action associée au signal ;
- `struct sigaction *oldact` : sauvegarde de l'ancienne action associée au signal (0 si on ne souhaite rien sauvegarder).

La structure `sigaction` contient les champs suivants :

- `void (*sa_handler)(int)` : pointeur sur la fonction à exécuter pour la gestion du signal ;
- `void (*sa_sigaction)(int, siginfo_t *, void*)` : pointeur sur la fonction à exécuter pour la gestion du signal si SA_SIGINFO est spécifié dans sa_flags ;
- `sigset_t sa_mask` : ensemble des signaux à bloquer pendant l'exécution de la fonction pointée par `sa_handler` ;
- `int sa_flags` : options de comportement du *handler*. 0 pour les options par défaut.

sigaction

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

Rappel du plan

int pause() définie dans unistd.h : le processus s'endort jusqu'à ce qu'un signal soit reçu.

unsigned int alarm(**unsigned int** nb_sec) définie dans unistd.h : arme une temporisation qui délivre le signal SIGALRM à son issue. alarm(0) annule l'alarme. Permet de clore un processus si un temps d'attente (ouverture d'un fichier, écriture dans un pipe ...) est jugé trop long.

Résumé

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

int kill(pid_t pid, int sig);
int sigwait (const sigset_t *set, int *sig);
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
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);
```

Remarque :

```
/* A sigset_t has a bit for each signal.  */
```

Exemple

*** block_ctrlc.c ***

Un programme qui bloque SIGINT (signal généré par CTRL+C) pour permettre à un calcul de continuer sans être interrompu.
Bloquer temporairement des signaux sert à prévenir des interruptions dans les parties critique d'un code : si un tel signal arrive, il est mis en attente et sera délivré plus tard (lorsque le signal sera débloqué).

**** block_ctrlc.c ****

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_INT 5000

int main(int argc, char *argv[])
{
    sigset_t signals1, signals2;
    unsigned int i = 0;

    sigemptyset(&signals1);        //empty signals1
    sigaddset(&signals1,SIGINT);   //select SIGINT
    sigprocmask(SIG_SETMASK,&signals1,0); //block the signals selected by signals1

    printf("Starting to count - cannot block me with CTRL+C\n");
    while(i < MAX_INT)
    {
        printf("\r%d going to %d, can't stop me  ",MAX_INT,i);
        usleep(1000); //wait 1 ms
        i = i+1;
    }

    sigpending(&signals2);      //retrieve pending signals
    if(sigismember(&signals2,SIGINT))
        printf("\nSIGINT is pending : 'CTRL C' was pressed\n");

    sigemptyset(&signals1);     //empty signals1
    sigprocmask(SIG_SETMASK,&signals1,0); //unblock SIGINT

    printf("\nSIGINT unblocked and not used.\n");
    return 0;
}
```

Sortie si on a appuyé sur CTRL+C pendant l'exécution :

```
$ ./block
Starting to count - cannot block me with CTRL+C
5000 going to 4999, can't stop me
SIGINT is pending : 'CTRL C' was pressed
```

Sortie si on n'a pas appuyé sur CTRL+C :

```
$ ./block
Starting to count - cannot block me with CTRL+C
5000 going to 4999, can't stop me
SIGINT unblocked and not used.
```

Exercice

***** new_handler.c *****

Écrire un programme qui définie un handler pour un certain signal : intercepter un segmentation faults (signal SIGSEGV), et quand trop de segfaults sont générés, envoyer un signal SIGKILL signal pour terminer le programme.

Sortie :

```
$ ./handler
Segmentation fault n. 1 (signal 11)
Segmentation fault n. 2 (signal 11)
Segmentation fault n. 3 (signal 11)
Segmentation fault n. 4 (signal 11)
Segmentation fault n. 5 (signal 11)
Segmentation fault n. 6 (signal 11)
Killed
```