II. Les signaux

1. Principe

Les signaux sont un mécanisme asynchrone de communication inter-processus. Intuitivement, ils sont comparables à des sonneries, les différentes sonneries indiquant des évènements différents. Les signaux sont envoyés à un ou plusieurs processus. Ce signal est en général associé à un évènement.

Le processus visé reçoit le signal sous forme d'un drapeau positionné dans son bloc de contrôle. Le processus est interrompu et réalise éventuellement un traitement de ce signal (routine correspondant au signal).

On peut considérer les signaux comme **des interruptions logicielles**, ils interrompent le flot normal d'un processus mais ne sont pas traités de façon synchrone comme les interruptions matérielles.

L'émission du signal est **volontaire** et donc parfaitement prévisible dans le déroulement chronologique du processus émetteur : c'est pour lui un **événement synchrone**. En revanche, la réception de ce signal par le processus récepteur peut arriver à n'importe quel moment, c'est donc un **événement asynchrone**. Il est même possible que le processus auquel le signal était destiné soit terminé.

2. Signaux sous UNIX

Un *signal* est la notification à un processus de l'occurrence d'un événement.
Les signaux sont envoyés :

- par un processus utilisateur à un autre processus;
- par un processus utilisateur à un groupe de processus
- par le noyau (Système) à un ou plusieurs processus.
- par le terminal (Clavier)

Les signaux peuvent être :

- certains caractères tapés au clavier:
  - interruption (Ctl-c) -> SIGINT
  - terminaison (Ctl-d) -> SIGQUIT
  - suspension (Ctl-z) -> SIGTSTP
- conditions hardware, ex:
  - erreur virgule flottante -> SIGFPE
  - adresse invalide -> SIGSEGV
- conditions software
  SIGILL : Instruction illégale

Chaque signal à un nom symbolique. La liste est définie dans <*sys/signal.h*>.
Cette liste peut être affichée à l'aide de la commande Unix : *kill –l*

Numéro	Nom	Description
1	SIGHUP	Instruction (HANG UP) - Fin de session
2	SIGINT	**Interruption depuis le clavier (Ctl-C)**
3	SIGQUIT	**Terminaison depuis le clavier(QUIT) (Ctl-D)**
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace trap
6	SIGABRT (ANSI)	Instruction (ABORT)
6	SIGIOT (BSD)	IOT Trap
7	SIGBUS	Bus error
8	SIGFPE	Floating-point exception - Exception arithmétique
9	SIGKILL	Instruction (KILL) - termine le processus immédiatement
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Violation de mémoire
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Broken PIPE - Erreur PIPE sans lecteur
14	SIGALRM	Alarme horloge
15	SIGTERM	Signal de terminaison
16	SIGSTKFLT	Stack Fault
17	SIGCHLD ou SIGCLD	modification du statut d'un processus fils
18	SIGCONT	Demande de reprise du processus
19	SIGSTOP	Demande de suspension logicielle
20	SIGTSTP	**Demande de suspension depuis le clavier (Ctl-Z)**
21	SIGTTIN	lecture terminal en arrière-plan
22	SIGTTOU	écriture terminal en arrière-plan
23	SIGURG	évènement urgent sur socket
24	SIGXCPU	temps maximum CPU écoulé
25	SIGXFSZ	taille maximale de fichier atteinte
26	SIGVTALRM	alarme horloge virtuelle
27	SIGPROF	Profiling alarm clock
28	SIGWINCH	changement de taille de fenêtre
29	SIGPOLL (System V)	occurence d'un évènement attendu
29	SIGIO (BSD)	I/O possible actuellement
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système
31	SIGUNUSED	

3. L'envoi de signaux : la primitive kill

Un signal est envoyé à un processus en utilisant l'appel système:

kill(int pid, int signal)

- signal est un numéro compris entre 1 et NSIG (défini dans <signal.h>)
- pid le numéro du processus.

```
Appel Système de la bibliothèque : #include <signal.h>
```

Il y a NSIG signaux sur une machine, déclarés dans le fichier.

La valeur de pid indique le PID du processus auquel le signal est envoyé :

- **pid > 0** : au processus identifié par pid

- 0 : à tous les processus du **groupe** du processus réalisant l'appel kill
- pid==-1. Dans ce cas, le signal est délivré à tous les processus qui ont les permissions suffisantes pour leur envoyer un signal.
- pid<-1. Dans ce cas, le signal est délivré à tous les processus qui font partie du groupe valeur absolue du (pid).

Le paramètre signal est interprété comme un signal si signal ∈[0-NSIG], ou comme une demande d'information si signal = 0 (suis-je autorisé à envoyer un signal à ce(s) processus ?). Comme un paramètre erroné sinon.

Le processus visé reçoit le signal sous forme d'un drapeau positionné dans son bloc de contrôle. Le processus est interrompu et réalise éventuellement un traitement de ce signal.
On peut considérer les signaux comme des interruptions logicielles, ils interrompent le flot normal d'un processus mais ne sont pas traités de façon synchrone comme les interruptions matérielles.

4. Redéfinition du Comportement d'un Signal :

L'appel système signal() permet à un processus de spécifier comment un signal particulier doit être traité. Il prend deux arguments : le numéro du signal dont le traitement doit être modifié et la fonction à exécuter lorsque ce signal est reçu.

```
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int Signum, sighandler_t Handler);
```

Signum : qui représente le premier argument de l'appel système *signal(2)* est généralement spécifié en utilisant les constantes définies dans signal.h (le numéro de signal).

Handler : est le second argument qui représente un pointeur vers une fonction de type void* qui prend comme argument un entier.

Fonctionnement :

La fonction signal permet de spécifier ou de connaître le comportement du processus à la réception d'un signal donné, il faut donner en paramètre à la fonction le numéro du signal sig que l'on veut détourner et la fonction de traitement Handler action à réaliser à la réception du signal.
Trois possibilités pour ce paramètre action

SIG_DFL : Comportement par défaut, plusieurs possibilités :

SIG_IGN : Le signal est ignoré
Remarque: les signaux SIGKILL, SIGSTOP ne peuvent pas être ignorés.

Handler : Une fonction définie par l'utilisateur.

Exemple :

On redéfini le comportement d'un signal passé en paramètre par le nouveau comportement (fonction *handler) qui* permet d'afficher le numéro du signal.

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

void  Afficher(int sig)
{
        printf(″ le numéro de signal : %d\n ″, sig)
}


main(int argc, char *argv[])
 {
   int pid sig ;
   pid = getpid() ;
/* on récupère le numéro de signal passé en paramètre */
   sig = atoi(argv[1]);
/* on redéfini le comportement du signal */
   signal(sig, Affichage) ;
/* on envoi le signal au processus en cours*/
   kill(pid, sig) ;

/*Boucle infinie*/
while(1);
}
```

A l'exécution le programme affiche le numéro du signal reçu avant de rentrer dans une boucle infinie à l'exception du signal SIGKILL et SIGSTOP.

5. Fonctions de mise en attente de réception d'un signal

pause(void) *:* cette primitive est le standard UNIX d'attente. Elle bloque le processus qui fait l'appel jusqu'a de la réception d'un signal quelconque.

int sigsuspend(cont sigset_t *p_ens) : mise en sommeil jusqu'a l'arrivée d'un signal non masqué appartenant à l'ensemble p_ens.

6. Synchronisation avec Signaux

On désire synchroniser l'exécution des processus en utilisant les signaux.

Exemple 1 :

Deux processus 1 et 2 s'exécutent en parallèle et qui réalisent deux traitements spécifiques A1 et A2.
On veut réaliser la synchronisation suivante :
Le processus 2 ne peut exécuter son traitement A2 que si le Processus 1 a terminé l'exécution de son traitement A1.

Processus 1	**Processus 2**
.	.
.	.
.	.
/* Traitement A1*/	/* Traitement A2*/
.	.
.	.
.	.

Solution :

- Bloquer le processus 2 avant d'entrer dans le traitement A2
- Réveiller le processus 2 après la fin de l'exécution du processus 1

Processus 1	**Processus 2**
.	.
.	**pid2= getpid();**
.	**pause()**
/* Traitement A1*/	/* Traitement A2*/
kill(pid2, Signal)	.
.	.
.	.

Signal : SIGCONT ou n'importe quel signal

Contrainte : le processus1 doit connaitre le pid du processus 2
Dans ce cas on lance le processus 2 en premier, on récupère son pid et on lance le processus 1 en passant en paramètre le pid du processus 2.

Exemple 2 :

Deux processus 1 et 2 s'exécutent en parallèle et qui réalisent deux traitements spécifiques A1, B1 pour le processus 1 et A2, B2 pour le processus 2.

On veut réaliser la synchronisation suivante : **A1; A2 ; B1; B2**

Processus 1	**Processus 2**
.	.
.	.
.	.
/* Traitement A1*/	/* Traitement A2*/
.	.
.	.
.	.
/* Traitement B1*/	/* Traitement B2*/
.	.
.	.
.	.

Principe de Solution :

- Bloquer le traitement en attente (Appel système pause())
- Débloquer un processus par l'envoi d'un signal (Appel système kill())

Processus 1	**Processus 2**
.	.
.	.
pid1= getpid();	**pid2= getpid();**
.	**pause()** ;
/* Traitement A1*/	/* Traitement A2*/
kill(pid2, Signal) ;	**kill(pid1, Signal) ;**
pause() ;	**pause() ;**
/* Traitement B1*/	/* Traitement B2*/
kill(pid2, Signal)	.

Contrainte :

- le processus 1 doit connaitre le pid du processus 2
- le processus 2 doit connaitre le pid du processus 1

Cas où les processus ont un lien de parenté

Le processus 1 et le processus 2 ont un lien de parenté par exemple le processus 1 est le père du processus 2.

```
                        pid = fork()
if  (pid > 0)                          If  (pid==0)

/* Processus 1 : Père */               /* Processus 2 : Fils */
        .                                      .
        .                                      .
        .                                   pause() ;
/* Traitement A1*/                     /* Traitement A2*/
      kill(pid, Signal) ;                  kill(getppid(), Signal) ;
      pause() ;                            pause() ;
        .                                      .
/* Traitement B1*/                      /* Traitement B2*/
       kill(pid, Signal)
```

Complément sur les Signaux

Le bloc de contrôle d'un processus contient un masque. Il définit l'ensemble des signaux dont la délivrance n'est pas assurée (blocage). Un processus peut installer un masque de signaux quelconque à l'exclusion de SIGKILL, SIGSTOP et SIGCONT

Manipulation des ensembles de signaux :

Le type sigset_t représente les ensembles de signaux.

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

int sigemptyset(sigset_t *p_ens) ;	***p_ens={}**
int sigfillset(sigset_t *p_ens) ;	***p_ens={1,…,NSIG}**
int sigaddset(sigset_t *p_ens,int sig); *	**p_ens=*p_ens U {sig}**
int sigdelset(sigset_t *p_ens,int sig);	***p_ens=*p_ens-{sig}**
int sigismember(sigset_t *p_ens,int sig);	**sig est dans *p_ens**

Chacune de ces fonctions renvoie –1 en cas d'erreur, 0 sinon
La fonction sigismember renvoie 0 ou 1 selon que sig appartient ou non à *p_ens.

Blocage des signaux

L'installation d'un masque de blocage des signaux est réalisée par un appel à la fonction :

int sigpromask(int **op**, const sigset_t ***p_ens**, sigset_t ***p_ens_ancien**)

Le nouveau masque est construit à partir de l'ensemble *p_ens et du masque antérieur *p_ens_ancien

Le paramètre op permet de déterminer le nouvel ensemble :

Valeur de op	**Nouveau masque**
SIGSET_TMASK	*p_ens
SIG_BLOCK	*p_ens_ancien U *p_ens
SIG_UNBLOCK	*p_ens_ancien - *p_ens

La valeur de retour de la fonction est 0 en cas de réussite, -1 sinon.

Liste des signaux pendant bloqués :

int sigpending(sigset_t ***p_ens**);

Un appel à la fonction sigpending écrit à l'adresse p_ens la liste des signaux pendants qui sont bloqués**.**

Manipulation des handlers :

La fonction signal : signal(sig, handler) présente deux inconvénients :

- A la délivrance du signal sig capté par le pid, le handler par défaut de sig est automatiquement réinstallé.
- L'interface signal ne permet pas de masquer aucun siganl pendant l'exécution du handler

La structure sigaction

```
Struct sigaction {
void (*sa_handler)() ;
sigset_t sa_mask ; /* signaux  à bloquer*/
int sa_flags ; /*options*/
}
```

Le champ sa_mask correspond à une liste de signaux qui doivent être ajoutés, pendant l'exécution du handler à ceux déjà bloqués.

Parmi les valeurs de sa_flags :
SA_RESTART **:** un appel système interrompu par un signal capté est repris au lieu de renvoyer –1
SA_RESETHAND **:** si le signal est capté, il ne sera pas bloqué à sa délivrance et SIG_DFL sera Réinstallé automatiquement

La primitive sigaction

int sigaction(int **sig**, const struct sigaction ***p_action**, struct sigaction ***p_action_ancien**)

Si p_action n'est pas NULL, alors cette primitive permet d'installer la fonction p_action->sa_handler comme handler pour le signal sig.

Les signaux p_action-> sa_mask U {sig} sont masqués pendant l'exécution du handler.