CHAPITRE 2 : Synchronisation des processus par les signaux

1- Concepts de base

1.1- Définitions

Dans le système UNIX (ou Linux) les signaux constituent le mécanisme de base de **synchronisation** des processus entre eux mais aussi entre le noyau et les processus.

Un signal est une **interruption logicielle** qui permet de notifier à un processus, de manière **asynchrone**, l'occurrence de l'événement associé à ce signal.

Cet événement peut être :

- une exception provoquée par le processus lui-même, dans ce cas c'est le noyau qui enverra au processus le signal associé à cette exception.
- un message de synchronisation émis par un autre processus, sous la forme d'un signal spécifique.

Dans le processus, la réception d'un signal est matérialisée par la mise à 1 d'un indicateur binaire associé à ce signal.

Dans le processus, l'ensemble de ces indicateurs représentant les différents signaux mis en œuvre par le système, sont regroupés dans un tableau de bits.
Un tableau parallèle contient les adresses des sous-programmes de **service** à exécuter lorsque le signal correspondant est reçu.

1.2- Les différents types de signaux

La commande **man 7 signal** permet de lister la description des différents signaux mis en œuvre par le système.

Chaque signal est identifié par un **nom symbolique** et un **numéro** (entier positif**)** qui correspond à son indice dans le tableau d'indicateurs défini ci-dessus.

A chaque signal est associé un traitement par **défaut** (sous-programme codé dans le noyau) qui sera exécuté par le processus lors de la réception du signal, à moins qu'un traitement de **substitution** n'ait été mis en place par le programmeur dans le processus.

Le traitements par défaut n'est pas le même pour tous les signaux et correspond à un des traitements suivant :

- **arrêt** du processus.
- **arrêt** du processus mais en plus le système sauve dans un fichier **core** « l'image mémoire du processus ».
- **Suspension** de l'exécution du processus.
- quand le processus reçoit le signal il l'**ignore.**

Remarques :

- Le traitement par défaut des signaux **SIGKILL** et **SIGSTOP** ne peut pas être modifié.
- Les types de signaux et leur numéro d'identification sont légèrement différents dans les distributions UNIX ou Linux. Il est donc conseillé d'utiliser le **nom symbolique** du signal plutôt que son **numéro.**

1.3- Description des signaux

Exemple des signaux utilisés sous UNIX SystemV

Name	Value	Default	Event
SIGHUP	1	Exit	Hangup [see termio(7)]
SIGINT	2	Exit	Interrupt [see termio(7)]
SIGQUIT	3	Core	Quit [see termio(7)]
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status Changed
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	33	Ignore	Urgent Socket Condition
SIGPOLL	22	Exit	Pollable Event [see streamio(7)]
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user) [see termio(7)]
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input) [see termio(7)]
SIGTTOU	27	Stop	Stopped (tty output) [see termio(7)]
SIGVTALRM	37	Exit	Virtual Timer Expired
SIGPROF	38	Exit	Profiling Timer Expired
SIGXCPU	35	Core	CPU time limit exceeded [see getriimit(2)]
SIGXFSZ	36	Core	File size limit exceeded [see getriimit(2)]
SIGIO	34	Core	Socket I/O possible

Exemple des signaux utilisés sous différents systèmes Unix/Linux

DESCRIPTION
Linux supports the signals listed below. Several signal numbers are architecture dependent.
First the signals described in POSIX.1.

Signal	Value	Action	Comment
SIGHUP	1	A	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	C	Quit from keyboard
SIGILL	4	C	Illegal Instruction
SIGABRT	6	C	Abort signal from abort(3)
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe: write to pipe with no readers
SIGALRM	14	A	Timer signal from alarm(2)
SIGTERM	15	A	Termination signal
SIGUSR1	30,10,16	A	User-defined signal 1
SIGUSR2	31,12,17	A	User-defined signal 2
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	DEF	Stop process
SIGTSTP	18,20,24	D	Stop typed at tty
SIGTTIN	21,21,26	D	tty input for background process
SIGTTOU	22,22,27	D	tty output for background process

Next various other signals.

Signal	Value	Action	Comment
SIGIOT	6	C	IOT trap. A synonym for SIGABRT
SIGEMT	7,-,7		
SIGSTKFLT	-,16,-	A	Stack fault on coprocessor
SIGIO	23,29,22	A	I/O now possible (4.2 BSD)
SIGCLD	-,-,18	A	synonym for SIGCHLD
SIGPWR	29,30,19	A	Power failure (System V)
SIGINFO	29,-,-	A	synonym for SIGPWR
SIGLOST	-,-,-	A	File lock lost
SIGWINCH	28,28,20	B	Window resize signal (4.3 BSD, Sun)
SIGUNUSED	-,31,-	A	Unused signal (will be SIGSYS)

(Here - denotes that a signal is absent; there where three values are given, the first one is usually valid for alpha and sparc, the middle one for i386 and ppc and sh, the last one for mips. Signal 29 is SIGINFO/SIGPWR on an alpha but SIGLOST on a sparc.)

The letters in the "Action" column have the following meanings:

A Default action is to terminate the process.

B Default action is to ignore the signal.

C Default action is to terminate the process and dump core.

D Default action is to stop the process.

E Signal cannot be caught.

F Signal cannot be ignored.

Next the signals not in POSIX.1 but described in SUSv2.

Signal	Value	Action	Comment
SIGBUS	10,7,10	C	Bus error (bad memory access)
SIGPOLL		A	Pollable event (Sys V). Synonym of SIGIO
SIGPROF	27,27,29	A	Profiling timer expired
SIGSYS	12,-,12	C	Bad argument to routine (SVID)
SIGTRAP	5	C	Trace/breakpoint trap
SIGURG	16,23,21	B	Urgent condition on socket (4.2 BSD)
SIGVTALRM	26,26,28	A	Virtual alarm clock (4.2 BSD)
SIGXCPU	24,24,30	C	CPU time limit exceeded (4.2 BSD)
SIGXFSZ	25,25,31	C	File size limit exceeded (4.2 BSD)

(For the cases SIGSYS, SIGXCPU, SIGXFSZ, and on some architectures also SIGBUS,
the Linux default action up to now (2.3.27) is A (terminate), while SUSv2 prescribes C (terminate and dump core).)

2- Émission d'un signal

Un signal peut être émis par un processus ou par le noyau lui même à destination d'un ou de plusieurs processus. L'émission d'un signal se traduit simplement par la mise à **1** du bit correspondant dans le tableau de bits de chacun des processus destinataires.

2.1- Émission par le noyau

Lorsque se produit une **exception** du type :

- interruption matérielle
- instruction illégale
- violation d'un segment mémoire
- erreur bus
- division par zéro
- etc …

le noyau envoie le signal correspondant au processus concerné.

2.2- Émission par un processus

La fonction **kill()** permet à un processus d'envoyer un signal à un processus ou à un groupe de processus (excepté ceux du **groupe 0** dont le propriétaire est **root**) :

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

        int  kill( pid_t id, int sig ) ;
```

-sig est le nom symbolique du signal émis

-id désigne le ou les processus destinataires :

- si *id* > 0 le signal est envoyé au processus de PID = *id*
- si *id* < -1 le signal est envoyé à tous les processus du groupe dont le GPID = | *id* |
- si *id* = 0 le signal est envoyé aux autres processus du groupe auquel appartient le processus émetteur
- si *id* = -1 le signal est envoyé à tous les processus dont le propriétaires **réel** est le **même** que le propriétaire **effectif** du processus émetteur.
  Si le processus émetteur appartient à **root** tous les processus reçoivent le signal sauf **init**.

Un processus n'a la permission d'envoyer un signal à d'autres processus que dans les conditions suivantes :

- ses propriétaires **réel** ou **effectif** sont les **mêmes** que ceux des processus destinataires
- son propriétaire **effectif** est **root** (EUID = 0)

La fonction retourne **–1** en cas d'erreur et 0 sinon.

Certaines versions récentes d'UNIX fournissent une nouvelle fonction **sigsend()** qui est plus simple d'utilisation que **kill()** :

```
#include <sys/types.h>
#include <signal.h>
#include <procset.h>

        int  sigsend ( idtype_t typeDest, id_t id, int sig ) ;
```

-sig est le nom symbolique du signal émis

- *typeDest* et *id* spécifient les processus destinataires de la façon suivante :
  - si *typeDest* = P_PID le signal est envoyé au processus de PID = *id*
  - si *typeDest* = P_PGID le signal est envoyé au groupe de processus de GPID = *id*
  - si *typeDest* = P_UID le signal est envoyé à tous les processus dont le propriétaire effectif à l'EUID = *id*
  - si *typeDest* = P_GID le signal est envoyé à tous les processus dont le groupe propriétaire effectif à l'EGID = *id*
  - si *typeDest* = P_ALL le signal est envoyé à tous les processus (excepté ceux du groupe 0)

Les permissions et le code de retour sont les mêmes que ceux de la fonction **kill().**

3- Réception d'un signal

La réception d'un signal par un processus consiste à tester régulièrement le tableau des indicateurs défini au § 1.1 et à exécuter le traitement de service associé à chaque indicateur positionné à **1**.

Un processus qui reçoit un signal peut réagir de différentes façons :

- il s'interrompt pour exécuter le traitement par **défaut**
- il s'interrompt pour exécuter un traitement **spécifique** prévu par le programmeur puis reprend son cours ou se termine
- il **ignore** le signal

Remarques :

Un inconvénient majeur du mécanisme des signaux est que le processus ne peut pas connaître l'identité du processus qui a émis le signal.

Un processus fils crée par un **fork()**, hérite des dispositions de traitements des différents signaux existantes dans le processus père.

Un processus qui change son code par un appel à **exec()**, réinitialise tous les traitements des signaux aux valeurs par défaut définies au **§ 1.3**.

Un processus peut modifier le traitement par défaut associé à un signal (sauf pour **SIGKILL** et **SIGSTOP**) en appelant l'une des deux fonctions **signal()** ou **sigset()**.

La fonction **signal()** permet de redéfinir le traitement associé à la réception d'un signal :**.**

#include <signal.h>

typedef void (* PtrFct) (int) ;

PtrFct **signal** (int *sig*, PtrFct *traitSig*);

- *sig* est le nom du signal dont on veut modifier le traitement
- *traitSig* est le nom de la fonction qui sera exécutée lors de la réception du signal et qui peut prendre une des valeurs suivantes :
  - **SIG_IGN** le signal est **ignoré**
  - **SIG_DFL** le traitement par **défau**t sera exécuté
  - *traitSig* le traitement **programmé** dans la fonction C *traitSig* sera exécuté

La fonction **signal** retourne une des valeurs suivantes :

- **SIG_ERR** en cas d'erreur
- **SIG_HOLD** si le signal est masqué
- un pointeur sur la fonction de traitement précédemment affectée au signal

Remarques :

- Le traitement par défaut des signaux SIGKILL et SIGSTOP ne peut pas être modifié.
- Si durant l'exécution du traitement d'un signal, une **nouvelle occurrence** de ce même signal se produit, elle ne sera prise en compte que lorsque le traitement en cours sera terminé.
- La fonction **signal()** implémentée dans les versions d'UNIX SystemV, a un fonctionnement légèrement différent. Dans ce cas, il est préférable d'utiliser la fonction **sigset()**.

Politique de traitement des signaux

La fonction *traitSig* passée en paramètre de la fonction **signal**, admet un paramètre d'entrée de type **int** qui sera le numéro du signal transmis à la fonction lors de son appel consécutif à la réception du signal :

void *traitSig* (int *sig*)
{
…… ;
}

On peut mettre en œuvre 2 types de fonctions de traitement :

- Une fonction qui traite **un signal particulier**. Dans ce cas le paramètre *sig* n'a pas d'utilité et peut être omis puisque la fonction est appelée uniquement lorsque ce signal est reçu par le processus.
- Une fonction qui traite **plusieurs signaux**. Dans ce cas la fonction est appelée chaque fois qu'un de ces signaux est reçu. Le paramètre *sig* doit alors servir d'aiguillage pour sélectionner le traitement associé au signal reçu.

Exemple1 :

```
/* cas d'une fonction qui traite uniquement le signal SIGUSR1*/
/* chaque fois que le processus recevra le signal SIGUSR1 */
/* il executera la fonction traiterUSR1( ) */

#include <sys/types.h>
#include <signal.h>

typedef void ( * PtrFct) (int) ;

void traiterUSR1( int ) ;

int main( )
{
…….
PtrFct  retFct ;    /* declaration d'un pointeur de fonction */
…….

/* mise en place du traitement (en general en tete du main) */

retFct = signal ( SIGUSR1, traiterUSR1);
if (retFct == SIG_ERR )
        { perror ("erreur signal") ; exit(1) ; }

……..

}
```

```
void traiterUSR1( int idSig)
{
……..
/* traitement de service */
……..
}
```

Remarques :

- normalement lorsque le traitement de service est terminé, le processus continue son exécution au point où il l'avait interrompue lors de la réception du signal SIGUSR1.
- si l'on veut arrêter le processus, il faut terminer le traitement de service par un **exit**.
- si l'on veut retourner vers un point de reprise, il faut terminer le traitement de service par un **longjmp**.

Exemple2 :

```
/* cas d'une fonction qui traite les signaux SIGUSR1 et SIGUSR2*/
/* chaque fois que le processus recevra un de ces 2 signaux */
/* il executera la fonction traiterUSRx( ) */

#include <sys/types.h>
#include <signal.h>

typedef void ( * PtrFct) (int) ;

void traiterUSRx( int ) ;

int main( )
{

…….
PtrFct  retFct ;    /* declaration d'un pointeur de fonction */
…….

/* mise en place du traitement (en general en tete du main) */
retFct = signal ( SIGUSR1, traiterUSRx) ;
if (retFct == SIG_ERR )
        { perror ("erreur signal") ; exit(1) ; }
retFct = signal ( SIGUSR2, traiterUSRx) ;
if (retFct == SIG_ERR )
        { perror ("erreur signal") ; exit(2) ; }
```

```
……..
}

void traiterUSRx( int  idSig)
{
…….
switch  ( idSig )
     {
     case  SIGUSR1 : /* traitement de service de SIGUSR1 */
                     …… ; break ;

     case  SIGUSR2 : /* traitement de service de SIGUSR2 */
                     …… ; break ;
     }
}
```

Masquage d'un signal

Un signal peut être « **masqué** » temporairement.
Cela consiste à mettre en place un mécanisme qui fait que lorsque le signal est reçu par le processus, il est mémorisé mais le traitement associé n'est pas exécuté tant que le masque est positionné.

Les fonctions suivantes permettent de gérer le masquage des signaux :

int **sighold** (int *sig*) ; cette fonction masque le signal *sig*

int **sigrelse** (int *sig*) ; cette fonction démasque le signal *sig*

Remarques :

- Les signaux SIGKILL et SIGSTOP ne peuvent pas être masqués.
- Les fonctions **signal** sous Linux ou **sigset** sous Unix systemV, font que lorsqu'un signal est reçu, le noyau **masque automatiquement** ce signal avant d'appeler le traitement de service. Si le traitement de service se termine normalement sans arrêter le processus, le noyau **démasque** le signal et le processus continue son exécution au point où il l'avait interrompue lors de la réception du signal.

- En conséquent, si le traitement de service d'un signal se termine par un **longjmp** (retour vers un point de reprise) penser à mettre un **sigrelse** juste avant le **longjmp** pour démasquer le signal qui a été masqué par le noyau lors de sa réception.

4- Autres primitives

4.1 Mise en sommeil d'un processus

Un processus peut se mettre « **en sommeil** » pendant *n* secondes, en appelant la fonction suivante :

```
#include <unistd.h>

        unsigned int  sleep ( unsigned int n ) ;
```

4.2 Attente d'un signal

Un processus peut se mettre « **en sommeil** » jusqu'à ce qu'il reçoive un signal, en appelant la fonction suivante :

```
#include <unistd.h>

        int  pause ( void ) ;
```

le processus est réactivé dés qu'il reçoit un signal **quelconque** non masqué et ensuite il se déroute dans le traitement de service de ce signal.
L'instruction qui suit **pause** ne sera exécutée que si le traitement de service ne tue pas le processus ou si ce traitement n'est pas terminé par un **longjmp**.

4.3- Gestion de délais

Un processus peut déclencher un compte à rebours (**timeout**) avec la fonction **alarm()** et se mettre « **en sommeil** » jusqu'à ce que le délai expire :

#include <unistd.h>

unsigned int **alarm** (unsigned int *durée*) ;

cette fonction initialise une horloge qui enverra le signal **SIGALRM** au processus dés que le délai *durée* (secondes) sera expiré.
Si *durée* est égal à **0** l'horloge est remise à 0 mais le signal **SIGALRM** n'est pas émis.
Pour arrêter l'horloge avant que le délai expire, il suffit donc de faire **alarm(0)** .