Chapitre 03 : Programmation des Signaux Systèmes UNIX/Linux

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Master MS2E

ECOLE SUPERIEURE POLYTECHNIQUE

www.esp.sn

Plan du Chapitre

Sommaire

Les signaux UNIX

- Un signal est une interruption logicielle qui est envoyée aux processus par le système pour les informer sur des événements anormaux se déroulant dans leur environnement
- Il permet également aux processus de communiquer entre eux
- A une exception près (SIGKILL), un signal peut être traité de trois manières différentes :
  1. Il peut être **ignoré**
     - le programme peut ignorer les interruptions clavier générées par l'utilisateur. C'est ce qui se passe lorsqu'un processus est lancé en arrière-plan
  2. Il peut **être pris en compte**
     - l'exécution d'un processus est détournée vers une procédure spécifiée par l'utilisateur, puis reprend où elle à été interrompue
  3. Son comportement par défaut peut être restitué par un processus après réception de ce signal

Sommaire

Les types de signaux UNIX

Description

- Signaux numérotés de 1 à 64 :
  ↪ 1 à 32 : signaux dits “classiques”
  ↪ 33 à 64 : signaux dits “temps réel”
- Associés à des constantes :
  ↪ Préfixe `SIG` suivi du nom
  ↪ Ne pas utiliser directement les numéros !
- En C : constantes donc majuscules
- Sous le terminal : en minuscules ou majuscules (ou les deux)
  ↪ Dépend de l’implémentation

Attention

La description et/ou la définition de certains signaux n’est pas POSIX !

Liste des signaux et description (1/2)

Description

Numéro	Nom	Description
1	SIGUP	Terminaison du processus leader de la session
2	SIGINT	Interruption du clavier (CTRL+C)
3	SIGQUIT	Caractère QUIT depuis le clavier (CTRL+\)
4	SIGILL	Instruction illégale
5	SIGTRAP	Point d'arrêt pour le débogage
6	SIGABRT	Terminaison anormale
7	SIGBUS	Erreur de bus
8	SIGFPE	Erreur mathématique virgule flottante
9	SIGKILL	Terminaison forcée du processus
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Accès mémoire invalide
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Écriture dans un tube sans lecteur
14	SIGALRM	Fin de temporisation alarme
15	SIGTERM	Terminaison du processus
16	SIGSTKFLT	Erreur de pile du coprocesseur

Liste des signaux et description (2/2)

Description

Numéro	Nom	Description
17	SIGCHLD	Terminaison du processus fils
18	SIGCONT	Reprise de l'exécution d'un processus stoppé
19	SIGSTOP	Stoppe l'exécution d'un processus
20	SIGSTP	Caractère suspend depuis le clavier (CTRL+Z)
21	SIGTTIN	Lecture par un processus en arrière-plan
22	SIGTTOU	Écriture par un processus en arrière-plan
23	SIGURG	Données urgentes dans une socket
24	SIGXCPU	Limite de temps processus dépassée
25	SIGXFSZ	Limite de taille de fichier dépassée
26	SIGVALRM	Alarme virtuelle
27	SIGPROF	Alarme du profileur
28	SIGWINCH	Fenêtre redimensionnée
29	SIGIO	Arrivée de caractères à lire
30	SIGPOLL	Équivalent à SIGIO
31	SIGPWR	Chute d'alimentation
32	SIGUNUSED	Signal non utilisé

Actions par défaut

Description

Nom des signaux	Action par défaut
SIGHUP, SIGINT, SIGBUS, SIGKILL, SIGUSR1, SIGUSR2, SIGPIPE, SIGALARM, SIGTERM, SIGSTKFLT, SIGXCOU, SIGXFSZ, SIGVTALARM, SIGPROF, SIGIO, SIGPOLL, SIGPWR, SIGUNUSED	Abandon
SIGQUIT, SIGILL, SIGTRAP, SIGABRT, SIGIOT, SIGFPE, SIG-SEGV	Abandon + fichier *core*
SIGCHLD, SIGURG, SIGWINCH	Ignoré
SIGSTOP, SIGSTP, SIGTTIN, SIGTTOU	Processus stoppé
SIGCONT	Processus redémarré

Sommaire

Réception d'un signal : Primitive signal()

Primitive signal()

- Valeur retournée: adresse de la fonction spécifiant le comportement du processus vis-à-vis du signal considéré
- -1 sinon
- fcn est un pointeur sur une fonction prenant un entier en paramètre.

```
1 #include <signal.h>
2
3 void (*signal (int sig, void (*fcn)(int)))(int)
4
5 /* reception d un signal */
6 /* sig : numero du signal */
7 /* (*fcn) : action apres reception */
```

- Cette primitive intercepte le signal de numéro sig
- Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes

Réception d'un signal : Primitive signal()

```
1 #include <signal.h>
2
3 void (*signal (int sig, void (*fcn)(int)))(int)
4
5 /* reception d un signal */
6 /* sig : numero du signal */
7 /* (*fcn) : action apres reception */
```

- Cette primitive intercepte le signal de numéro sig
- Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes
  1. SIG_DFL : ceci choisit l'action par défaut pour le signal. La réception d'un signal par un processus entraîne alors la terminaison de ce processus, sauf pour SIGCLD et SIGPWR, qui sont ignorés par défaut. Dans le cas de certains signaux, il y a création d'un fichier image core sur le disque.

Réception d'un signal : Primitive signal()

```
1 #include <signal.h>
2
3 void (*signal (int sig, void (*fcn)(int)))(int)
4
5 /* reception d un signal */
6 /* sig : numero du signal */
7 /* (*fcn) : action apres reception */
```

☛ Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes

2. SIG_IGN : ceci indique que le signal doit être ignoré : le processus est immunisé. On rappelle que le signal SIGKILL ne peut être ignoré.

Réception d'un signal : Primitive signal()

```
1 #include <signal.h>
2
3 void (*signal (int sig, void (*fcn)(int)))(int)
4
5 /* reception d un signal */
6 /* sig : numero du signal */
7 /* (*fcn) : action apres reception */
```

- Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes
  3. Un pointeur sur une fonction (nom de la fonction) : ceci implique la capture du signal.
     - La fonction est appelée quand le signal arrive, et après son exécution, le traitement du processus reprend ou il à été interrompu
     - On ne peut procéder à un déroutement sur la réception d'un signal SIGKILL puisque ce signal n'est pas interceptable

Émission d'un signal : Primitive kill()

Primitive kill()

- ☛ Valeur retournée : 0 si le signal a été envoyé, -1 sinon.

```
1 #include <signal.h>
2 int kill(pid_t pid,int sig)
3 /* emission dun signal */
4 /* pid : identificateur du processus ou du groupe destinataire */
5 /* sig : numero du signal */
```

- ☛ Cette primitive émet à destination du processus de numéro pid le signal de numéro sig
- ☛ De plus, si l'entier sig est nul, aucun signal n'est envoyé, et la valeur de retour permet de savoir si le nombre pid est un numéro de processus ou non.
- ☛ Utilisation du paramètre pid :
  1. Si $pid > 0$: pid désigne alors le processus d'identificateur pid.

Émission d'un signal : Primitive kill()

```
1 #include <signal.h>
2 int kill(pid_t pid,int sig)
3 /* emission dun signal */
4 /* pid : identificateur du processus ou du groupe destinataire */
5 /* sig : numero du signal */
```

☛ Utilisation du paramètre pid :

1. Si $pid > 0$: pid désigne alors le processus d'identificateur pid.
2. Si $pid = 0$: le signal est envoyé à tous les processus du même groupe que l'émetteur
   - cette possibilité est souvent utilisée avec la commande shell (kill -9 0) pour tuer tous les processus en arrière plan sans avoir à indiquer leurs identificateurs de processus

Émission d'un signal : Primitive kill()

```
1 #include <signal.h>
2 int kill(pid_t pid,int sig)
3 /* emission dun signal */
4 /* pid : identificateur du processus ou du groupe destinataire */
5 /* sig : numero du signal */
```

☛ Utilisation du paramètre pid :

3 Si $pid = -1$:

- Si le processus appartient au super-utilisateur, le signal est envoyé à tous les processus, sauf aux processus système et au processus qui envoie le signal
- Sinon, le signal est envoyé à tous les processus dont l'identificateur d'utilisateur réel est égal à l'identificateur d'utilisateur effectif du processus qui envoie le signal
  - C'est un moyen de tuer tous les processus dont on est propriétaire, indépendamment du groupe de processus

4 Si $pid < -1$: le signal est envoyé à tous les processus dont l'identificateur de groupe de processus (pgid) est éegal à la valeur absolue de pid

Réception d'un signal : Primitive pause()

```
1 #include <unistd.h>
2 void pause(void)
3 /* attente d'un signal quelconque */
```

Primitive pause()

- Cette primitive correspond à de l'attente pure. Elle ne fait rien, et n'attend rien de particulier
- Cependant, puisque l'arrivée d'un signal interrompt toute primitive bloquée, on peut tout aussi bien dire que pause() attend un signal
- On observe alors le comportement de retour classique d'une primitive bloquée, c'est-à-dire le positionnement de errno à EINTR
- Notons que le plus souvent, le signal que pause() attend est l'horloge d'alarm().

Émission d'un signal : Primitive alarm()

Primitive alarm()

- Valeur retournée : temps restant dans l'horloge.

```
1 #include <unistd.h>
2 unsigned alarm(unsigned secs)
3 /* envoi d'un signal SIGALRM */
4 /* secs : nombre de secondes */
```

- Cette primitive envoie un signal SIGALRM au processus appelant après un laps de temps **secs** (en secondes) passé en argument, puis réinitialise l'horloge d'alarme.
- A l'appel de la primitive, l'horloge est initialisée à **secs** secondes, et est décrémentée jusqu'à 0. Si le paramètre secs est nul, toute requête est annulée
- Cette primitive peut être utilisée, par exemple, pour forcer la lecture au clavier dans un délai donné
- Le traitement du signal doit être prévu, sinon le processus est tué

Sommaire

Signal SIGCHLD : gestion des processus zombis

- Le signal SIGCHLD (anciennement SIGCLD) est un signal utilisé pour réveiller un processus dont un des fils vient de mourir
- C'est pourquoi il est traité différemment des autres signaux
- La réaction à la réception d'un signal SIGCHLD est de repositionner le bit pendant à zéro, et d'ignorer le signal
  - Mais le processus a quand même été réveillé pour cela.

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {

    if (fork() != 0) {
        while(1) ; /* boucle executée par le pére */
    }

    return 0;
}
```

Signal SIGCHLD : gestion des processus zombis

- ☛ L'effet d'un signal SIGCHLD est donc uniquement de réveiller un processus endormi en priorité interruptible.
- ☛ Si le processus capture les signaux SIGCHLD, il invoque alors la procédure de traitement définie par l'utilisateur comme il le fait pour les autres signaux, ceci en plus du traitement par défaut.
- ☛ Le traitement normal est lié à la primitive wait qui permet de récupérer la valeur de retour (exit status) d'un processus fils

```c
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    signal(SIGCLD,SIG_IGN) ;/* ignore le signal SIGCLD */
    if (fork() != 0) {
        while(1);
    } return 0;
}
```

Signal SIGCHLD : gestion des processus zombis

☛ En effet, la primitive wait est bloquante et c'est la réception du signal qui va réveiller le processus, et permettre la fin de l'exécution de la primitive wait.

```c
#include <signal.h>
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    signal(SIGCLD,SIG_IGN) ;/* ignore le signal SIGCLD */
    if (fork() != 0) {
        while(1);
    } return 0;
}
```

Sommaire

Exercice 1 : Communication entre processus

```
 1 #include <stdio.h>
 2 #include <errno.h>
 3 #include <signal.h>
 4 #include <unistd.h>
 5
 6 void it_fils()
 7 {
 8     printf("\t Interruption : kill (getpid(),SIGINT)\n") ;
 9     kill (getpid(),SIGINT) ;
10     }
11 void fils() {
12     signal(SIGUSR1,it_fils);
13     printf("Looping !\n");
14     while(1);
15     }
16
17 int main() {
18
19     int pid ;
20     if ((pid=fork())==0) fils() ;
21     else {
22
23         sleep(3) ;
24         printf(" kill (pid,SIGUSR1) ;\n") ;
25         kill (pid,SIGUSR1) ;
26     }
27     return 0;}
```

Exercice 2 : Émission d'un signal

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

void it_horloge(int sig){/* routine executee sur reception de SIGALRM */

    printf("Reception du signal %d : SIGALRM \n",sig); }

int main() {

    unsigned sec;

    signal(SIGALRM,it_horloge); /* interception du signal */

    printf("On fait alarm(5)\n") ;

    sec=alarm(5) ;

    printf("Valeur retournee par alarm : %d\n",sec) ;

    printf("Le principal boucle Ã  l'infini (CTRL-c pour arrÃªter)\n") ;

    for(;;);
    return 0; }
```

Exercice 3: programme qui attend des signaux (1/2)

```c
#include <errno.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int cpt=0;
void handler(int signum) {
    if(signum == SIGUSR1) {
    printf("Signal 1 recu\n");cpt = cpt + 1; }
    if(signum == SIGUSR2) {
        printf("Signal 2 recu\n");cpt = cpt + 2; }
}
int main() {
    if(signal(SIGUSR1, handler) == SIG_ERR) {
        perror("Erreur lors du positionnement ");
        exit(EXIT_FAILURE);
    }
    if(signal(SIGUSR2, handler) == SIG_ERR) {
        perror("Erreur lors du positionnement ");
        exit(EXIT_FAILURE);
    }
    printf("Pret a recevoir des signaux. Mon PID : %d\n", getpid());
    while(cpt != 3){
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Exercice 3 : programme qui envoie des signaux (2/2)

```c
#include <errno.h>
#include <signal.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define _POSIX_SOURCE
int main(int argc, char *argv[]) {
    pid_t pidServeur;
    if(argc != 2) {
        fprintf (stderr,"Tapez %s pid\n",argv[0]); exit(EXIT_FAILURE) ;
    }

    pidServeur = atoi(argv[1]);

    printf ("Attente avant envoi premier signal\n") ; sleep (1) ;
    printf ("Envoi premier signal\n") ;

    if (kill(pidServeur, SIGUSR1) == 1 ){
        perror("Erreur lors de l'envoi du signal "); exit(EXIT_FAILURE);
    }

    printf("Attente avant envoi deuxieme signal\n"); sleep(2);
    printf( "Envoi deuxieme signal\n" ) ;

    if(kill(pidServeur, SIGUSR2 ) ==   1 ) {
        perror("Erreur lors de l'envoi du signal "); exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Chapitre 03 : Programmation des Signaux Systèmes UNIX/Linux

Dr Mandicou BA
mandicou.ba@esp.sn
http://www.mandicouba.net

Master MS2E

ECOLE SUPERIEURE POLYTECHNIQUE

www.esp.sn