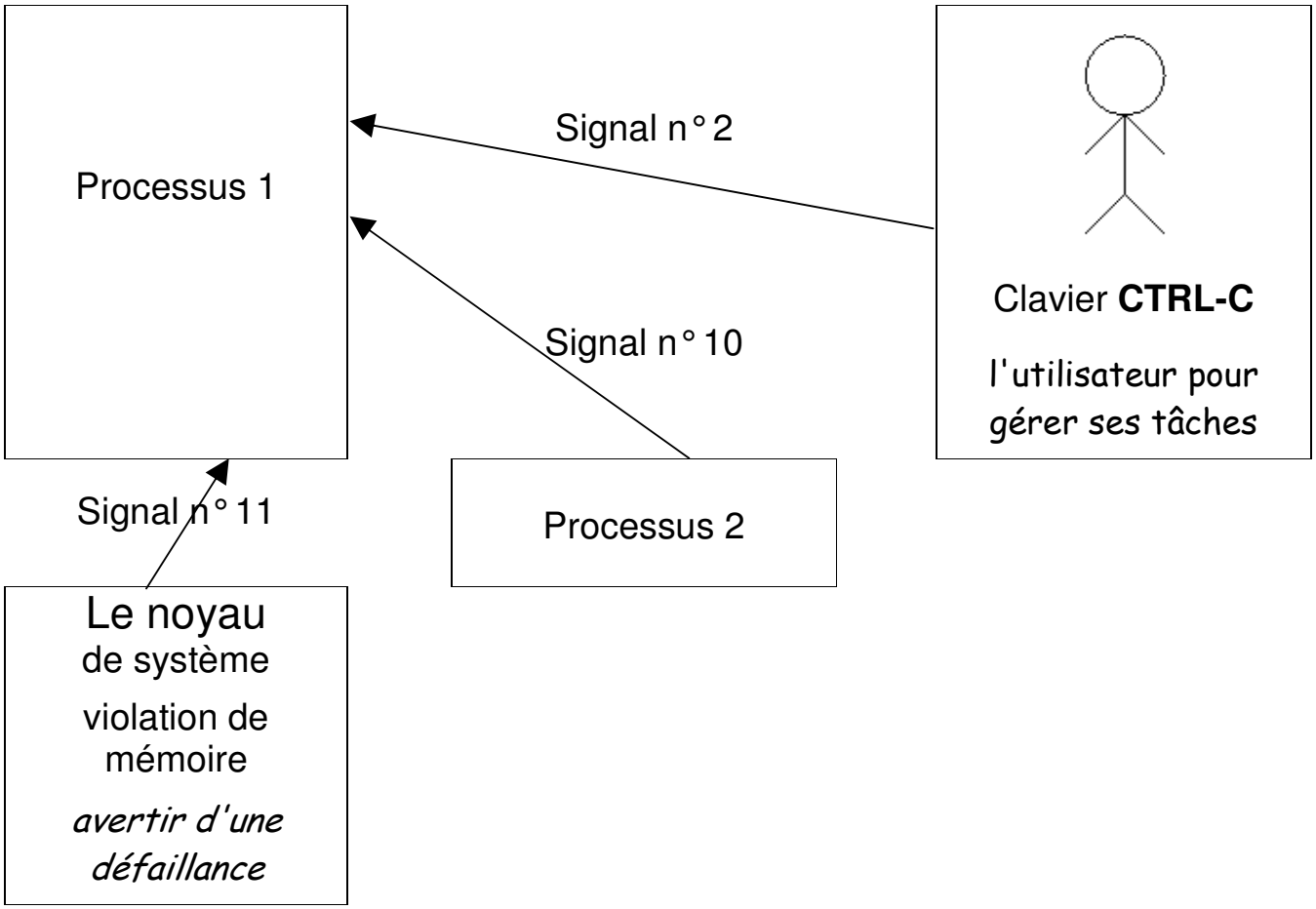
Chapitre 2

Communication interprocessus (suit)

2.3/ Communication par signaux.

2.3.1/ Introduction :

Les signaux (ou les interruptions) servent à informer les processus de l'arrivée d'un événement et quelque chose s'est produit dans le système, on lui associe également l'action à entreprendre :



Références :
L. Bobelin Unix, polycopiés de cours, Polytech, 2011
J. Boukachour, Polycopiés de cours, Université du Havre, 2013.

Les signaux de Unix ont des origines diverses,

les signaux :

- Peuvent être transmis à tout moment ou bien occasionnellement

- Sont émis à **partir :**
  - **du clavier**(suite à la frappe d'une combinaison de touches : par exemple **CTRL-C** pour envoyer le signal SIGINT; CTRL D...)

  - d'un **processus**(par la primitive kill dans un programme C).

  - par la commande **kill** depuis le **shell**

  - ou du **noyau**(lors de la constations d'une anomalie matérielle : Violation mémoire, erreur E/S, division par zéro)

- Permettent d'informer les processus de l'occurrence d'un événement exceptionnel, jamais comme un moyen de communication ordinaire

2.3.2/ Comportement du processus (Réaction aux signaux)

Lors de la réception d'un signal, il y a des actions par défaut suivant le signal :

- Ignorer le signal sans exécuter aucun traitement
- Terminer le programme.
- Un processus peut changer son comportement par défaut lors de la réception d'un signal en indiquant la fonction à exécuter lors de la réception du signal.
- Avec les signaux il n'y a pas de communication de données. Ils sont identifiés par un numéro entier et un nom symbolique décrit dans signal.h.
- Le récepteur ne peut pas déterminer l'identité de l'émetteur.
- L'interruption d'un processus par un signal est **impossible** quand il est en **mode noyau** : Il faut qu'il passe en mode utilisateur.
- Les signaux sont sans effet sur un processus zombi

2.3.3/ Identification des signaux

Unix gère 64 signaux différents, Chaque signal est identifié par un nom et un numéro. On distingue deux classes de signaux :

– Classiques, numérotés de 1 à 31
– Signaux temps réels, numérotés de 32 à 63

```
$ 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
```

Voici la signification des signaux principaux :

- SIGINT 2 signal d'interruption (provoqué par CTRL-C par exemple) ;
- SIGQUIT 3 caractère quit frapper au clavier (Ctrl - \) ;
- ...
- SIGKILL 9 fin du processus (non déroutable) ;
- SIGUSR1 10 (aussi 30,16) signal émis par un processus utilisateur (disponible pour les applications)
- SIGSEGV 11 violation d'espace mémoire
- SIGUSR2 12 (aussi 31,17) signal émis par un processus utilisateur (disponible pour les applications)
- SIGPIPE 13 écriture dans un tube sans lecteur
- SIGALRM 14 signal déclenché après un certain nombre de secondes (horloge) ;
- SIGTERM 15 fin de processus ;
- ...

- SIGCONT 18 est envoyé à un processus pour lui faire reprendre sonexécution (après un SIGSTOP).
- SIGSTOP 19 (CNTL-Z) pour pouvoir stopper un processus (stopper pour reprendre plus tard, pas arrêter) ;
- ...
- SIGIO 29
- SIGUSR1 30 signal utilisateur 1 (disponible pour les applications) ;
- SIGUSR2 31 signal utilisateur 2 (disponible pour les applications).

- Certains signaux ont des statuts particuliers :

    - SIGKILL ne peut pas être intercepté, bloqué ou ignoré. Cela permet de tuer un signal même en cas de processus récalcitrant.

    - SIGSTOP est dans le même cas, pour pouvoir stopper un processus (stopper pour reprendre plus tard, pas arrêter).

    - SIGCONT il est envoyé à un processus pour lui faire reprendre son exécution (après un SIGSTOP). Il est donc logique qu'il ne puisse être pris en charge par un handler .

2.3.4/ Comment manipuler les signaux

Dans le cas du système Unix/Linux, un processus utilisateur peut **envoyer** un signal à un autre processus.

- Les deux processus appartiennent au même propriétaire,
- ou bien le processus émetteur du signal est le super-utilisateur.

Il existe 2 moyens :

- Par la ligne de commande,
- Par une primitive en c.

Le programmeur peut mettre en place des méthodes (handlers) pour **prendre en charge** chacun des signaux reçus.

2.3.4.1/ Commande Kill (shell)

- La commande kill permet d'envoyer un signal à un processus
- **Syntaxe du kill :** Kill -n°du signal (ou le nom du signal) PID
- **Exemples du Kill :**

  `Kill -9 2235` (envoie du signal 9 au processus 2235)

  le processus recevant ce signal exécutera la fonction correspondante (au signal 9) qui est toujours la terminaison (état zombie)

  `Kill -2 2235` (envoie du signal 2 au processus 2235) (équivalent de CTRL-C)

- Le signal SIGKILL (N°9) ne pas être dérouté. Le comportement par défaut à la réception du signal 9 est la destruction du processus.
- Si le numéro du signal n'est pas précisé, kill envoie le signal 15 (SIGTERM)
- La commande `kill -l` donne la liste des signaux d'un système Unix.

2.3.4.2/ La primitive kill (en C) :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

Il prend en argument le `pid` du processus destinataire du signal et le numéro du signal à envoyer `signal`.

`pid` peut prendre comme valeurs :
. Si pid > 0 : processus pid.
. Si pid = 0 : groupe de l'émetteur.
. Si pid = -1 : tous les processus (seulement root peut le faire).

La valeur de retour de `kill` est 0 en cas de succès ou -1 en cas d'échec.

Dans le bloc de contrôle d'un processus, le noyau mémorise entre autres :
– Les signaux reçus
– Les fonctions de traitement des signaux

La réponse d'un processus à un signal déclenche l'exécution d'une routine standard.

Il est possible de modifier la réponse en définissant une fonction spécifique, appelée `handler`

ATTENTION : cet appel système est particulièrement mal nommé (kill : tuer), car il ne tue que très rarement un processus

Exemple d'envoi d'un signal

```
/* sig1.c */
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
void main(void) {
pid_t p;
int etat;
if ((p=fork()) == 0) { /* processus fils qui boucle */
while (1);
exit(2);
}
/* processus pere */
sleep(6);
printf("envoi de SIGUSR1 au fils %d\n", p);
kill(p, SIGUSR1);
// bloque l'appelant (pere) et selectionne le fils
p = waitpid(p, &etat, 0);
printf("etat du fils %d : %d\n", p, etat);
}
```

Voici le résultat de l'exécution :

```
cc  sig1.c  -o  sig1

./sig1  &
[1] 9211
nakechbm@corton:~/s4#/tp5$ envoi de SIGUSR1 au fils 9212
etat du fils 9212 : 10

[1]+  Termine 23              ./sig1
```

2.3.5/ Traitement des signaux

Méthodes de traitement :

- Traitement standard

- Traitement par une fonction définie par l'utilisateur (un handler).

Règles générales :

- A chaque type de signal est associé un handler par défaut. Le plus commun est la terminaison du programme (kill -9).

- Un processus peut ignorer un signal en lui associant le handler,

- si le processus exécute un programme utilisateur : traitement immédiat du signal reçu,

- s'il se trouve dans une fonction du système (ou system call) : le traitement du signal est différé jusqu'à ce qu'il revienne en mode user, c'est à dire lorsqu'il sort de cette fonction.

2.3.5.1 Traitement standard du signal.

Deux stratégies de gestion des signaux :

- SIG_DFL traitement du signal par défaut
- SIF_IGN signal est ignoré

Exemple 1 : Ci-dessous un programme non interruptible avec CTRL-C ; SIGINT est ignoré. *Le programme doit être arrêté avec kill -9 no.PID. Par contre, si l'appel signal() est mis en commentaire, le programme s'arrête bien avec CTRL-C.*

```
// sig3_1.c
#include<signal.h>
void main (){signal (SIGINT,SIG_IGN) ;
for ( ; ;) printf(" boucle infinie \n"); }
```

Exemple 2:

```
/* sig3_2.c */
#include <stdio.h>
#include <signal.h>
int main(void)
{    signal(SIGINT, SIG_DFL);
/* traitement standard  du signal  SIGINT   */
    printf(" Dans le programme principal (man) :\n ");
  printf("appui sur Ctrl-C pour envoyer le signal n° 2 : \n");
  printf("ce signal va provoquer le traitement \n
standard : signal(SIGINT, SIG_DFL) \n");
    printf(" qui arrêtera le processus \n");
    printf(" Avant la boucle :\n ");
    for (;;) { }
    printf(" apres la boucle :\n ");return 0;}
```

Voici le résultat de l'exécution :

```
./sig3_2
 Dans le programme principal (man) :
 appui sur Ctrl-C pour envoyer le signal n° 2 :
ce signal va provoquer le traitement
 standard : signal(SIGINT, SIG_DFL)
 qui arrêtera le processus
 Avant la boucle :
^C
```

2.3.5.2 Mise en place d'un handler

Un handler est la fonction qui décrit la suite des instructions à effectuer lors de la réception d'un signal.

Les signaux (autres que SIGKILL, SIGCONT et SIGSTOP) peuvent avoir un handler spécifique installé par un processus :

La primitive `signal()`

```
#include<signal.h>
signal(numero_signal, methode_de_traitement);
```

- Elle installe le handler spécifié par `methode_de_traitement` pour le signal `numero_signal`.

- La fonction de handler (**`methode_de_traitement`**) est exécutée par le processus à la délivrance d'un signal (**`numero_signal`**). A la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.

Exemples de mise en place handler

Exemple1 : de mise en place handler

```
/* sig4_1.c */
#include <stdio.h>
#include <signal.h>
void traitement_interruption(int signum)
{
printf("Procédure simulant le traitement d'interruption
\n");}
int main(void)
{    printf(" Dans le programme principal (man) \n ");
    printf("appui sur Ctrl-C pour envoyer le signal n°
    2 : \n");
    signal(SIGINT, traitement_interruption);
    printf(" avant la boucle :\n ");
    for (;;) { }
    printf(" après la boucle :\n ");
    return 0;}
```

Voici le résultat de l'exécution :

```
cc  sig4_1.c  -o  sig4_1
./sig4_1
 Dans le programme principal (man) :
  avant la boucle :
^C appui sur Ctrl-C
Arret au prochain coup
^C
```

Exemple2 : Mise en place handler : Déroutement

Ci-dessous un programme provocant une division par zéro :

```
/*sig4_2.c */
#include<stdlib.h>
#include<stdio.h>
#include <unistd.h>
#include <signal.h>
main() { int a, b, Resultat;
         printf("Taper a : "); scanf("%d", &a);
         printf("Taper b : "); scanf("%d", &b);
         Resultat = a/b;
        printf("La division de a par b = %d\n", Resultat);}
```

Voici le résultat de l'exécution :

```
./sig4_2
Taper a : 5
Taper b : 0
Exception en point flottant
```

Nous allons mettre en place un handler permettant un traitement personnalisé de la division par zero :

```
/*sig4_3.c */
#include <signal.h>
#include <stdio.h>
void Hand_sigfpe() {
    printf("\nErreur division par 0 !\n");
    exit(1);
}
main() {
    int a, b, Resultat;
    signal(SIGFPE, Hand_sigfpe);
    printf("Taper a : "); scanf("%d", &a);
    printf("Taper b : "); scanf("%d", &b);
    Resultat = a/b;
    printf("La division de a par b = %d\n", Resultat);
}
```

Voici le résultat de l'exécution :

```
./sig4_3
Taper a : 5
Taper b : 0
Erreur division par 0 !
```