

Introduction aux Systèmes et Réseaux

TD n°2 : Réalisation des processus – Signaux

Le principaux points développés dans ce TD sont les suivants :

- compréhension des signaux.
- communication entre processus au moyen de signaux.
- utilisation élémentaire des signaux.

Les notions sur les signaux seront appliquées dans le TP n°2.

1 Signaux

1.1 Définition

Un signal est un message asynchrone destiné à un processus pour l'informer d'une situation particulière. Un processus qui reçoit un signal réagit en exécutant une action spécifiée, qui dépend de la nature du signal.

Un signal est analogue à une interruption ; mais alors qu'une interruption est destinée à un processeur physique, un signal est destiné à un processus. Certains signaux traduisent d'ailleurs directement l'occurrence d'une interruption matérielle.

Un signal peut être envoyé par le système d'exploitation (par exemple pour indiquer une erreur), ou bien par un processus.

Les signaux les plus courants sont :

- SIGINT (frappe du caractère *control-C*),
- SIGSTOP (signal de suspension d'un processus),
- SIGCONT (continuation d'un processus suspendu),
- SIGKILL (signal de terminaison)
- SIGCHLD (fin ou suspension d'un processus fils signalée à son père).

Voir `man 7 signal` pour une liste complète des signaux sous Unix.

1.2 Les signaux en attente

Un signal est *en attente/pendant* (*pending* en anglais) tant qu'il n'a pas été pris en compte par le processus destinataire. Il ne peut exister qu'un signal *pendant* d'un type donné par processus destinataire. Donc il est possible que des signaux soient perdus, par exemple si plusieurs signaux d'un certain type arrivent pendant le traitement d'un signal de ce même type.

1.3 Réception et traitement des signaux : les traitants *handlers*

Lorsqu'il reçoit un signal, un processus exécute une séquence de code (un *traitant*, ou *handler* en anglais) qui a auparavant été spécifiée pour le signal en question.

Tout signal a une action par défaut sur le processus destinataire (par exemple tuer le processus pour **SIGKILL**, suspendre le processus pour **SIGSTOP**, ne rien faire pour **SIGCHLD**, etc.). Il est néanmoins possible de spécifier un comportement particulier lors de la réception d'un signal. Cela peut se faire grâce à une primitive appelée **signal**, mais son emploi est déconseillé (car non compatible avec la norme POSIX¹). La primitive spécifiée par POSIX (**sigaction**) est néanmoins d'un maniement compliqué. C'est pourquoi on utilisera une primitive appelée **Signal**, fournie dans **csapp.c**, qui est une enveloppe (*wrapper*) de **sigaction** et offre une interface plus simple.

```
#include <signal.h>
typedef void handler_t(int);
handler_t *Signal(int signum, handler_t *handler);
```

Signal associe la fonction **handler** au signal **signum**. Notons que la fonction **handler** a une forme bien spécifiée : elle prend un entier en paramètre et ne renvoie rien.

Tous les comportements par défaut associés aux signaux peuvent ainsi être changés, sauf ceux des signaux **SIGKILL** et **SIGSTOP**.

1.4 Envoi de signaux

Les signaux peuvent être envoyés aux processus :

- à la suite d'une interruption (le système s'en charge)
- à la suite de la frappe d'une combinaison de touches (Ctrl-C, Ctrl-Z)
- en utilisant l'appel système *kill*²

La primitive *kill* peut être utilisée en ligne de commande

```
kill 12678 //arrêter le processus avec PID 12678, sinon kill -s <signal> <PID>
```

ou alors de manière programmatique

```
int kill(pid_t pid, int sig).
```

Pour la ligne de commande, le signal par défaut qui est envoyé est le signal d'arrêt (**SIGTERM**). Dans le cas programmatique, *kill* permet d'envoyer un signal **sig** à différents processus (si **pid** > 0, au processus **pid**, si **pid** = 0, aux processus du même groupe que l'appelant, si **pid** < -1 aux processus du groupe **|pid|**).

1. POSIX est une norme développée sous l'impulsion de l'IEEE, visant à uniformiser l'interface de différents systèmes d'exploitation. Il est recommandé de suivre cette norme, pour améliorer la portabilité des programmes.

2. Cette primitive est nommée **kill** pour des raisons historiques, mais son effet n'est pas nécessairement de tuer le processus destinataire.

Exercice 1 Que fait le programme suivant ?

```
#include "csapp.h"

void handler(int sig) /* SIGINT handler */
{
    printf("Caught SIGINT\n");
    exit(0);
}

int main()
{
    /* Install the SIGINT handler */
    Signal(SIGINT, handler);
    pause(); /* wait for the receipt of a signal */
    exit(0);
}
```

Exercice 2 De quel groupe sont les fils d'un processus ?

Exercice 3 Que fait le programme suivant ?

```
#include "csapp.h"
int main()
{
    int i;
    pid_t pid;

    for (i = 0; i < 2; i++)
        if ((pid = Fork()) == 0) { /* child */
            while(1);
        }

    Kill(-getpid(), SIGTERM);

    exit(0);
}
```

Exercice 4 Et celui-ci ?

```
#include "csapp.h"
int main()
{
    int i;
    pid_t pid;

    for (i = 0; i < 2; i++)
        if ((pid = Fork()) == 0) { /* child */
            while(1);
        }

    Kill(getpid(), SIGTERM);

    exit(0);
}
```

Exercice 5 La primitive `unsigned int sleep(unsigned int t)` suspend le processus appelant pendant `t` secondes ou jusqu'à l'arrivée d'un signal. Dans ce dernier cas, elle renvoie le nombre de secondes qui restaient à attendre.

Écrire un programme qui suspend le processus pendant un nombre de secondes passé en paramètre, et imprime le nombre de secondes effectivement passées à attendre, le programme pouvant être interrompu par *control-C*.

Exercice 6 La primitive `unsigned int alarm(unsigned int t)` provoque l'envoi d'un signal `SIGALRM` au processus appelant au bout de `t` secondes. Contrairement à `sleep`, `alarm` ne bloque pas le processus appelant.

Écrire un programme qui ne fait rien (exécute une boucle vide) mais qui reçoit un signal `SIGALRM` toutes les secondes, et affiche alors un message "bip". À la réception du sixième signal, le programme affiche "bye" et se termine.

Exercice 7 On considère le programme suivant :

```
#include "csapp.h"
int counter = 0;
void handler(int sig)
{
    counter++;
    sleep(1);
    return;
}

int main()
{
```

```

int i;
Signal(SIGUSR2, handler);

if (Fork() == 0) { /* child */
    for (i = 0; i < 5; i++) {
        Kill(getppid(), SIGUSR2);
        printf("sent SIGUSR2 to parent\n");
    }
    exit(0);
}

Wait(NULL);
printf("counter=%d\n", counter);
exit(0);
}

```

Que fait ce programme ?

Quelle est la valeur imprimée pour la variable `counter` ? Y-a-t-il un problème potentiel ?

Exercice 8 Lorsqu'un processus crée des processus fils, il peut attendre leur fin avec la primitive `wait` ou `waitpid`. Néanmoins, il ne peut pas faire de travail utile pendant cette attente. C'est pourquoi on souhaite que le processus père traite la fin de ses fils uniquement au moment où cette fin est signalée par le signal `SIGCHLD`.

Écrire le programme d'un processus créant plusieurs fils et traitant leur fin comme il vient d'être indiqué.

Attention : tenir compte du fait que les signaux ne sont pas mémorisés (un seul signal d'un type donné peut être dans l'état pendant, cf. question précédente).

Exercice 9 *BONUS*

Cette question est facultative et pourra être reportée à plus tard, mais elle sera utile pour des exercices ultérieurs.

Un processus peut bloquer la réception des signaux d'un certain type. Un signal bloqué parvient au processus destinataire, mais ne provoque aucun effet jusqu'à ce qu'il soit débloqué (c'est l'équivalent du masquage des interruptions matérielles).

Pour le masquage, il faut manipuler un tableau de bits qui représente le masque (un bit par signal, 1=bloqué, 0=non bloqué). Le type opaque `sigset_t` représente un tel tableau de bits pour l'ensemble des signaux. L'interface pour la manipulation de ce tableau est donnée ci-après (l'effet est indiqué en commentaire).

```

int sigemptyset(sigset_t *p_ens)          // *p_ens = ensemble vide
int sigfillset(sigset_t *p_ens)           // *p_ens = {1, ..., NSIG-1}
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 appartient à *p_ens?

```

Le masquage proprement dit est réalisé par la primitive :

```
int sigprocmask(int op, const sigset_t *p_ens, sigset_t *p_ensAncien);
```

qui modifie le masque du processus en fonction de la valeur de op.

- op=SIG_SETMASK : nouveau masque = *p_ens
- op=SIG_BLOCK : nouveau masque = masque courant U *p_ens
- op=SIG_UNBLOCK : nouveau masque = masque courant - *p_ens

Si *p_ensAncien est différent de NULL, alors l'ancienne valeur du masque est stockée dans cette variable.

La primitive `int sigpending(sigset_t *p_ens)` écrit dans *p_ens la liste des signaux pendants qui sont bloqués.

Écrire un programme qui place un masque sur 2 signaux (SIGINT et SIGUSR1), s'envoie ces signaux, imprime la liste des signaux pendants masqués, puis débloque les signaux masqués.

2 Pour les curieux (*bonus*) : Allocation du processeur

2.1 Rappel sur l'allocation de ressources

Rappelons qu'un processus représente l'exécution d'un programme. Pour faire exécuter un ensemble de processus sur un ordinateur, il faut partager la mémoire et le processeur entre ces processus. Pour cela, on utilise la notion de *ressource virtuelle* : une ressource virtuelle est une image d'une ressource réelle. Pour permettre l'exécution effective d'un processus, le système d'exploitation matérialise les ressources virtuelles par les ressources physiques correspondantes (on dit qu'il alloue les ressources physiques).

Dans un premier temps, nous considérons deux ressources : la mémoire et le processeur. On examinera plus tard les communications entre processus, les fichiers et les entrées-sorties.

- Chaque processus dispose d'une *mémoire virtuelle* dont la taille est égale à la capacité d'adressage du processeur, soit 2^{32} octets pour une taille d'adresse de 32 bits. C'est au système d'exploitation d'allouer au processus la mémoire physique correspondante, à mesure de ses besoins. Nous n'examinons pas ici les détails de la gestion de la mémoire virtuelle.
- Le processeur est alloué tour à tour aux processus. Pour respecter l'équité entre processus, chacun d'eux reçoit le processeur pendant une tranche de temps fixée appelée *quantum*, dont la durée est de l'ordre de 10 à 50 millisecondes. Néanmoins, lorsqu'un processus doit attendre (par exemple, attente de la réponse à une demande de lecture de fichier sur disque), il ne peut pas utiliser le processeur. Il doit alors le céder à un autre processus.

2.2 Détails sur l'allocation de processeur

C'est le système d'exploitation qui réalise l'allocation du processeur. Lorsque le processeur passe d'un processus au suivant (à la fin d'un quantum), le système doit sauvegarder l'état du processus interrompu (en particulier le contenu des registres) et recharger l'état

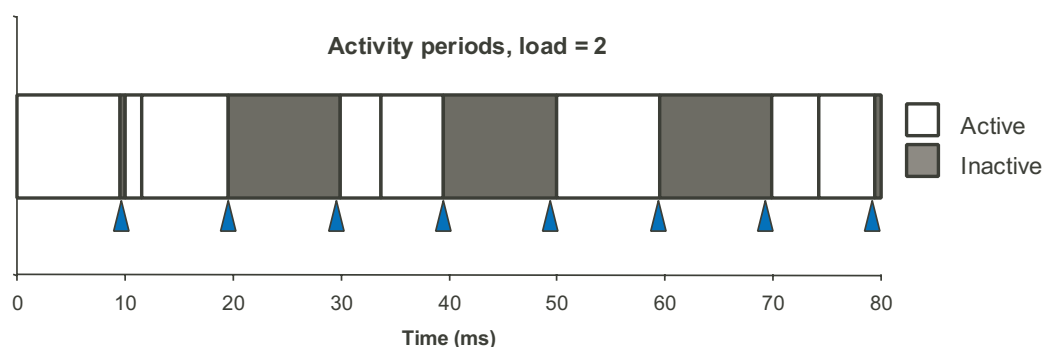
du nouveau processus. Ces opérations sont exécutées en mode superviseur, alors que les processus des usagers du système sont exécutés en mode utilisateur.

Il est possible de visualiser cette allocation du processeur en examinant le temps passé. En effet, le système permet de mesurer le temps d'exécution d'un processus, le temps réel écoulé, et le temps consommé par le système pour ses tâches de gestion.

À titre d'exemple, on considère le tableau ci-après, qui note le temps d'activité d'un processus particulier pendant une certaine période de temps. Les périodes d'activité (pour le processus considéré) sont notées A (par ex. A48, etc.) et les périodes d'inactivité sont notées I (par ex. I48, etc.). On note aussi la date de début et la durée de chaque période, en cycles d'horloge et en millisecondes.

A48	Time	191514104	(349,4 ms)	Duration	5224961	(9,532449 ms)
I48	Time	196739065	(358,93 ms)	Duration	247557	(0,451644 ms)
A49	Time	196986622	(359,38 ms)	Duration	858571	(1,566382 ms)
I49	Time	197845193	(360,95 ms)	Duration	8297	(0,015137 ms)
A50	Time	197853490	(360,97 ms)	Duration	4357437	(7,949733 ms)
I50	Time	202210927	(368,91 ms)	Duration	5718758	(10,433335 ms)
A51	Time	207929685	(379,35 ms)	Duration	2047118	(3,734774 ms)
I51	Time	209976803	(383,08 ms)	Duration	7153	(0,01305 ms)
A52	Time	209983956	(383,1 ms)	Duration	3170650	(5,784552 ms)
I52	Time	213154606	(388,88 ms)	Duration	5726129	(10,446783 ms)
A53	Time	218880735	(399,33 ms)	Duration	5217543	(9,518916 ms)
I53	Time	224098278	(408,85 ms)	Duration	5718135	(10,432199 ms)
A54	Time	229816413	(419,28 ms)	Duration	2359281	(4,304286 ms)
I54	Time	232175694	(423,58 ms)	Duration	7096	(0,012946 ms)
A55	Time	232182790	(423,6 ms)	Duration	2859227	(5,21639 ms)
I55	Time	235042017	(428,81 ms)	Duration	5718793	(10,433399 ms)

La figure ci-après est une autre représentation de cette trace :



Les triangles représentent les interruptions d'horloge (*timer*) qui provoquent le passage d'un processus à un autre. On notera que ce passage (réallocation du processeur) prend lui-même un certain temps. On notera aussi qu'il y a d'autres interruptions de très courte durée.

Exercice 10 Sur la trace ci-avant, indiquer si le processus observé était actif ou inactif

au moment de chacune des interruptions.

Exercice 11 Déterminer la période et la fréquence des interruptions d'horloge (*timer*), qui délimitent chaque quantum de temps.

Exercice 12 Déterminer la période et la fréquence d'horloge du processeur.

Exercice 13 Quelle est la durée du traitement des interruptions de l'horloge (appel à l'ordonnanceur) ?

Exercice 14 Expliquer pourquoi les plus longues périodes d'inactivité ont une durée supérieure à celle des plus longues périodes d'activité.

Question 15 On considère un ordinateur partagé par 100 utilisateurs, tous occupés à une tâche d'édition de textes. L'éditeur utilisé provoque une interruption lors de la frappe de chaque caractère. Le traitement de cette interruption consomme environ 100 000 cycles d'horloge. Les utilisateurs frappent en moyenne 100 mots par minute et un mot comporte en moyenne 6 caractères. La fréquence d'horloge de l'ordinateur est 1 GHz.

Estimer la fraction du temps total consommée par le traitement des interruptions. Le résultat obtenu vous paraît-il raisonnable ? (noter qu'il s'agit d'une limite supérieure largement surestimée, puisqu'on suppose que tous les utilisateurs frappent de manière continue à la cadence maximale).