Processus, partie 2

LIFSE - Systèmes d'exploitation

L. Gonnord, N. Louvet, G. Pichon, M. Moy, F. Zara

8 février 2024

Introduction

Le système sépare les processus pour éviter les perturbations, assurer la protection des données, pour faciliter la gestion.

Sans faire appel au système, un processus ne peut pas agir sur un autre.

Mais la communication entre processus est nécessaire

- certains processus doivent communiquer, *e.g.*, serveur
- tout processus nécessite un moyen d'être contacté, ne serait-ce que pour pouvoir l'arrêter !

Il existe différentes manières de faire communiquer des processus entre-eux : les paramètres de la fonction `main()`, les variables d'environnement, les signaux, les pipes...

Le principe

Dans les UE d'algorithmique et programmation, vous êtes habitués à écrire des programmes (non bogués) qui se terminent :

- soit quand l'utilisateur le demande via un menu ou une invite,
- soit tous seuls en quelques minutes au plus...

Mais il existe des applications qui :

- n'ont pas vocation à se terminer seules, et fonctionnent longtemps,
- qui n'interagissent pas avec un utilisateur dans un terminal.

Exemple : tout service réseau, comme un serveur web.

Comment faire pour interagir avec un processus sans utiliser de canal de communication de données (comme son entrée et sa sortie standard) ?

On se restreint des codes entiers, qui vont provoquer des actions simples : « peux-tu te terminer ? », « peux-tu effectuer tel traitement ? », ...

Le mécanisme utilisé est celui des signaux.
En général, quand un processus reçoit un signal :

- il est interrompu, et reçoit juste un code entier,
- en fonction du code reçu, il effectue un traitement,
- *il reprend son exécution là où il a été interrompu*.

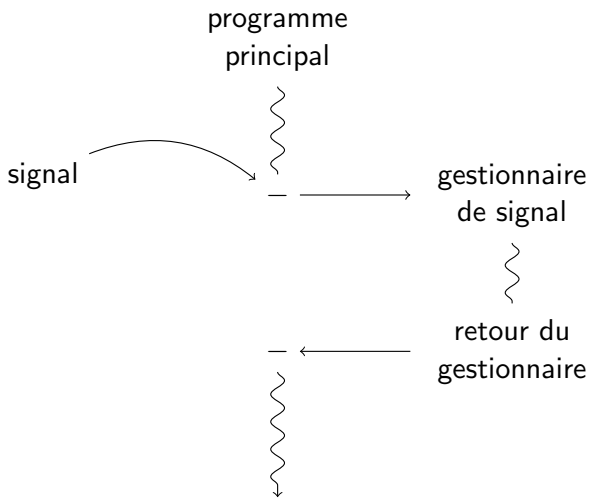
On parle de gestionnaire de signal pour désigner le traitement effectué lors de la réception d'un signal particulier.

À la réception d'un signal, on considère différents cas :

- le processus peut être terminé par le système,
- il peut suivre un comportement prédéfini (ex. sommeil),
- il peut exécuter un gestionnaire défini par le programmeur.
- Un processus peut aussi ignorer certains signaux. . .

Le cas où le signal termine le processus est plutôt simple : on s'intéresse aux cas des signaux qui ne causent pas la terminaison dans la suite.

Par exemple, supposons que le processus exécute son programme principal ; à la réception du signal, il passe à l'exécution du gestionnaire de signal :



Au retour du gestionnaire, l'exécution du programme reprend : tout se passe comme si une fonction avait été exécutée sans appel explicite.

Les signaux sont, pour le processus, des événements asynchrones : on ne sait pas *a priori* quand ils vont se produire.

Les signaux doivent vous faire penser aux interruptions : les mécanismes sont très proches, mais les signaux se veulent plus portables.

La réception d'un signal ne peut être prise en compte par un processus que lors de la reprise de son exécution après un passage en mode noyau : jamais « en plein milieu d'une instruction machine », mais une ligne de C/C++ peut se traduire en de nombreuses instructions machine !

Un exemple en pseudo code :

soit a une variable entière globale initialisée à 0
on installe un gestionnaire pour `SIGUSR1` qui effectue
    $a \leftarrow a - 1$
répéter 5 fois
    $a \leftarrow a + 1$
afficher a

On suppose que le processus reçoit une fois le signal `SIGUSR1` pendant l'exécution de la boucle : qu'affiche-t-il à la fin ?

Il faut être vigilant lorsqu'on modifie des variables partagées dans un gestionnaire de signal, même si c'est souvent nécessaire !

Dans l'exemple, l'affectation $\mathtt{a} \leftarrow \mathtt{a} + 1$ se décompose en trois instructions :

1 :	$\mathtt{R} \leftarrow \mathtt{a}$	{chargement de la mémoire vers un registre}
2 :	$\mathtt{R} \leftarrow \mathtt{R} + 1$	{en une instruction machine}
3 :	$\mathtt{a} \leftarrow \mathtt{R}$	{rangement d'un registre vers la mémoire}

- si le signal est pris en compte juste avant l'ins. 1, ou juste après l'ins. 3, a sera décrémentée par le gestionnaire ;
- si le signal est pris en compte entre l'ins. 1 et l'ins. 2, a ne sera pas décrémentée par le gestionnaire.

Dans un cas le programme affiche 4 et dans l'autre 5...

Il faut être vigilant lorsqu'on modifie des variables partagées dans un gestionnaire de signal, même si c'est souvent nécessaire !

D'autre part, que se passe-t-il si un processus reçoit un signal

- alors qu'il exécute le code d'une fonction ?
  ↪ rien de spécial ; après la gestion du signal, il reprend l'exécution normale de la fonction.
- alors qu'il exécute une primitive système ?
  ↪ après gestion du signal, l'appel système est interrompu !

Par exemple, si un processus reçoit un signal pendant qu'il lit $\texttt{n} > 0$ octets dans un fichier avec `read()`, alors l'appel se termine en retournant le nombre $\texttt{m}$ d'octets effectivement lus, et $\texttt{m} < \texttt{n}$.

Les appels systèmes, même bloquants, sont par défaut interrompus par la réception de signaux : il faut le gérer en se référent à leur documentation !

Gestion des signaux en C/C++

En C/C++, les numéros des signaux sont manipulés à l'aide de macros définies dans `signal.h`,

- `SIGINT` $\rightsquigarrow$ 2 (Ctrl+c),
- `SIGKILL` $\rightsquigarrow$ 9,
- `SIGUSR1` $\rightsquigarrow$ 10,
- `SIGTERM` $\rightsquigarrow$ 15,
- `SIGSCONT` $\rightsquigarrow$ 18,
- `SIGSTOP` $\rightsquigarrow$ 19 (Ctrl+z), ...

En effet, les codes peuvent changer d'un système à l'autre (ceux indiqués ici ne sont que des exemples), donc il vaut mieux utiliser les macros...

Pour envoyer le signal `signum` au processus dont l'identifiant est `pid`, on utilise la fonction suivante : `int kill(pid_t pid, int signum)`

Pour installer un gestionnaire de signal pour le signal `signum` :

```
int sigaction(int signum, const struct sigaction *new,
                                                  struct sigaction *old)
```

Lors d'un appel :

- la structure pointée par `new` décrit le gestionnaire que l'on veut installer pour le signal `signum` ;
- la structure pointée par `old` reçoit une copie du gestionnaire qui était installé jusqu'à présent pour `signum`.

`new` ou `old` peuvent être `NULL` :

- si `new` est `NULL`, on recopie simplement dans `old` le gestionnaire actuel ;
- si `old` est `NULL`, on installe `new` comme nouveau gestionnaire.

La structure `sigaction` est « intimidante » :

```
struct sigaction {
  void     (*sa_handler)(int);
  void     (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int      sa_flags;
  void     (*sa_restorer)(void);
};
```

Mais il ne faut pas se laisser impressionner ! On repart d'une structure existante, et on modifie juste le pointeur de fonction `sa_handler`.

Supposons par exemple que l'on veut qu'à la réception du signal `SIGINT` notre processus exécute une fonction d'en-tête `void fct(int)` :

```
struct sigaction s;
sigaction(SIGINT, NULL, &s);
s.sa_handler = fct;
sigaction(SIGINT, &s, NULL);
```

Primitives permettant d'attendre la réception d'un signal :

- `int pause(void)` : le processus appelant reste en sommeil jusqu'à ce qu'il reçoive un signal.
- `int sleep(unsigned int t)` : le processus appelant reste en sommeil jusqu'à ce qu'il reçoive un signal ou que t secondes soient écoulées.

Dans les deux cas, à la réception d'un signal, le gestionnaire de signal est exécuté, puis l'exécution reprend après l'appel à la primitive.

`unsigned int alarm(unsigned int t)` : le processus appelant recevra le signal `SIGALARM` `t` secondes après l'appel à la primitive.

Cas d'un processus père dont un fils se termine : lorsqu'un processus se termine, son père reçoit le signal `SIGCHLD`. Il peut alors prendre en compte la terminaison de son fils par un appel non bloquant à `waitpid()`.

Gestion des signaux en ligne de commande

Commandes shell importantes :

- `kill -l` : permet de lister les signaux utiliser sur le système.
- `kill -<SIGNAL> <PID>` : pour envoyer le signal `SIGNAL` au processus identifié par `PID`.

L'argument `SIGNAL` de `kill`, seulement en ligne de commande, peut être

- soit le nom complet du signal, par exemple `SIGKILL`,
- soit le nom abrégé du signal, par exemple `KILL`,
- soit le numéro du signal, par exemple 9.

Ainsi, pour tuer sauvagement le processus de PID 6543, vous pouvez utiliser `kill -SIGKILL 6543`, `kill -KILL 6543` ou encore `kill -9 6543`.

Un exemple

On veut écrire un processus qui, indéfiniment, affichage de `"RRRRR..."` puis se met en sommeil pendant 1 seconde. Lorsque le processus a reçu 5 fois le signal `SIGINT`, il se termine après avoir affiché `"Je me termine..."`.
On commence par écrire un pseudo-code :

soit `terminer` un drapeau global initialisé à faux
soit `cpt` un compteur global initialisé à 0 pour « compter `SIGINT` »
on installe un gestionnaire pour `SIGINT` dans lequel
    `cpt` est incrémenté
    si `cpt` atteint 5 alors `terminer` passe à vrai
dans le programme principal
    répéter indéfiniment
        afficher `"RRRRR..."`
        se mettre en sommeil 1 seconde
        si `terminer` alors sortir de la boucle
    afficher `"Je me termine..."`

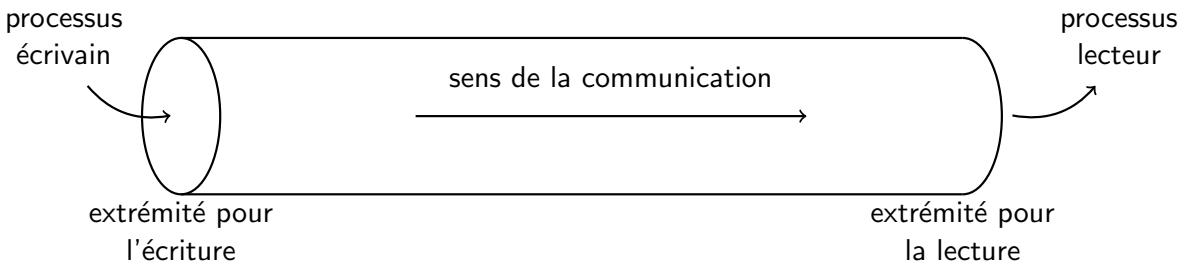
```
bool terminer = false; // drapeau de terminaison
int cpt = 0; // compte les signaux SIGINT reçus

void gest(int s) { // gestionnaire de signal
  cpt++;
  std::cout << "SIGINT reçu (cpt = " << cpt << ")" << std::endl;
  if(cpt == 5) terminer = true;
}

int main(void) {
  struct sigaction s;
  sigaction(SIGINT, NULL, &s);
  s.sa_handler = gest;
  sigaction(SIGINT, &s, NULL);

  while(true) {
    std::cout << "RRRRR..." << std::endl;
    sleep(1);
    if(terminer) break;
  }
  std::cout << "Je me termine..." << std::endl;
  return 0;
}
```

Le principe

Les tubes permettent de transmettre des données d'un processus à un autre :

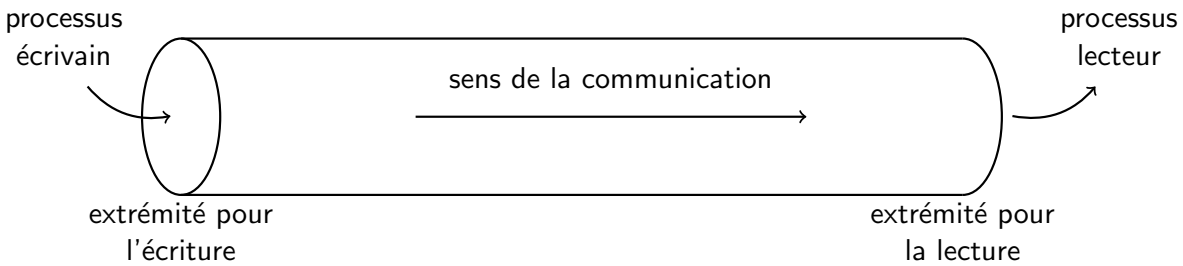
- la communication est unidirectionnelle,
- se fait sous la forme d'une séquence d'octets,
- se fait en mode First In First Out (FIFO).

Un processus appelé écrivain envoie des octets sur l'extémité du tube dédié à l'écriture, un autre appelé lecteur lit à l'autre extrémité.



L'envoi des séquences `abc` puis `def` est semblable à `abcdef`, et peut être lu en totalité ou par morceaux, comme `ab`, puis `cde`, puis `f` par exemple.

Pour fonctionner, un tube doit avoir au moins un écrivain et un lecteur :



Le tube a une capacité finie, et il y a une synchronisation de type producteur-consommateur entre le lecteur et l'écrivain :

1. le lecteur doit attendre qu'il y ait effectivement des données à lire dans le tube (ex : l'écrivain n'a pas encore écrit) ;
2. l'écrivain doit attendre qu'il y ait un lecteur et de la place dans le tube pour y écrire ses données (ex : lecteur pas assez rapide).

La capacité des tubes sous Linux est très grande (`man 7 pipe`), donc il n'est pas évident d'observer le cas 2 sur de petits exemples.

Synchronisation de type producteur-consommateur ?

Un producteur fait passer des données à un consommateur au travers d'une file de données de longueur bornée :

- le producteur produit des données à son rythme ; s'il n'y a plus de place dans la file pour envoyer les données produite, il attend.
- le consommateur les consomme à son rythme ; s'il n'y a pas de données disponibles en sortie de file, il attend qu'elles arrivent.

En d'autres termes,

- l'écrivain joue le rôle du producteur,
- le lecteur joue le rôle du consommateur.

Un tube est donc une file d'octets, dont les extrémités sont partagées entre deux processus. Ces extrémités sont manipulées comme des fichiers :

- l'extrémité pour l'écriture est écrite avec `write()`,
- l'extrémité pour la lecture est lue avec `read()`.

Tubes anonymes

Ce sont les fichiers spéciaux les plus simples pour la communication inter-processus sont les tubes anonymes ou pipes.

Les tubes fournissent un canal de communication unidirectionnel entre deux processus :

- ils ont une extrémité d'écriture et une de lecture ;
- ils ont une taille limitée et peuvent être remplis ;
- ils n'ont pas de nom et doivent donc être partagés après leur création.

extrait de `man pipe`

- `int pipe(int pipefd[2]);`
- `pipe()` creates a pair of file descriptors and places them in the array pointed to by `pipefd` :
  - `pipefd[0]` is for reading,
  - `pipefd[1]` is for writing.
- On success, zero is returned. On error, -1 is returned.

Pour échanger via un tube anonyme, deux processus doivent avoir un lien familial, et se le partager : un processus sera écrivain, l'autre sera lecteur.

Tant qu'un processus du système possède un descripteur de fichier en écriture sur le tube, tout appel à `read()` sur le tube est bloquant.

Un appel à `read()` sur un tube qui ne comporte plus d'écrivain retourne 0, ce qui indique que plus aucune donnée ne pourra être lue.

Lorsque l'écrivain a fini d'écrire, il ferme donc son descripteur en écriture, pour que le lecteur sache qu'il n'aura plus rien à lire, et se termine.

Si un processus tente d'écrire sur un pipe avec `write()` alors qu'il n'y a pas/plus de lecteur sur le pipe, cet écrivain reçoit le signal SIGPIPE ; par défaut, un processus qui reçoit le signal SIGPIPE se termine.

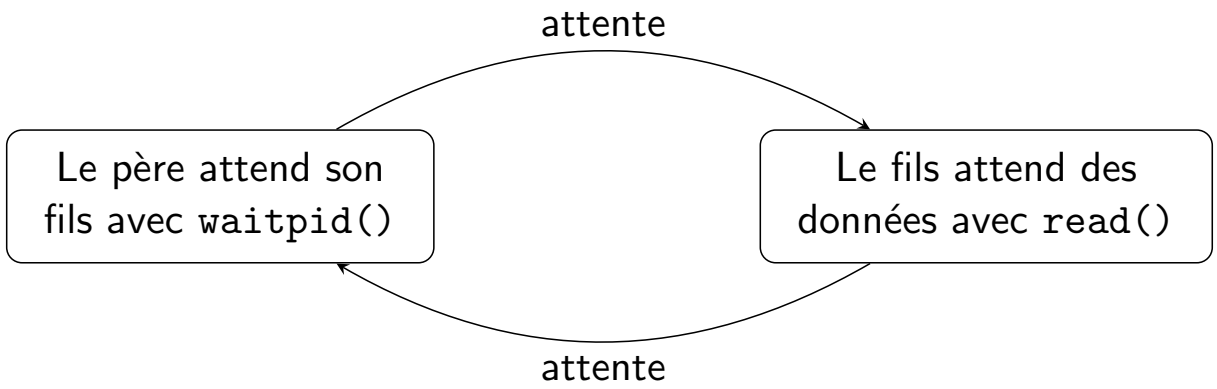
D'une façon générale, il est conseillé de fermer les descripteurs de fichiers qu'un processus n'utilise pas :

- le lecteur ferme le descripteur en écriture du pipe,
- l'écrivain ferme le descripteur en lecture du pipe.

Toujours libérer les ressources non utilisées !

Exemple : le fils attend de lire sur un pipe des octets en provenance de son père, et le père attend la fin de son fils avec `waitpid()` ; mais le père n'a pas fermé son extrémité du pipe en écriture...

Les deux processus sont bloqués, on est dans une situation d'interblocage :



Un exemple

On écrit un programme dans lequel le processus principal crée un pipe avec la fonction `pipe()`, puis un fils avec `fork()`.

Le processus principal joue le rôle d'écrivain, et va écrire un par un chaque caractère de l'alphabet de `'a'` à `'z'` :

- avec un petit message sur la sortie standard,
- sur l'entrée en écriture du pipe.

Ensuite le processus attend son fils puis se termine.

Le fils joue le rôle de lecteur : il lit un à un chaque caractère reçu sur le pipe, et l'affiche sur la sortie standard accompagné d'un petit message.

```
int main(void) {
    int ret, p[2]; // p[0] lecture, p[1] écriture
    pipe(p);
    ret = fork();
    if(ret > 0) { // processus père, écrivain
        close(p[0]);
        for(char c = 'a'; c <= 'z'; c++) {
            cout << "(père) j'écris " << c << endl;
            write(p[1], &c, 1);
        }
        close(p[1]);
        waitpid(ret, NULL, 0);
    }
    else { // processus fils, lecteur
        char c;
        int i = 0;
        close(p[1]);
        while(read(p[0], &c, 1) == 1)
            cout << "(fils) je lis " << c << endl;
        close(p[0]);
    }
    return 0;
}
```

Tubes nommés

Pour que deux processus sans lien familial puissent échanger via un tube, il faut qu'il ait un nom dans le système de fichiers : notion de tube nommé.

Un tube nommé ou fifo :

- est un tube qui a un nom dans le système de fichiers ;
- il est géré comme tout autre fichier :
  - emplacement dans le système de fichiers, droits,
  - ouverture avec `open()`, fermeture avec `close()`,
  - lecture avec `read()`, écriture avec `write()` ;
- c'est un tube, et une fois ouvert, il s'utilise comme tel.

Appel système : `int mkfifo(const char *name, mode_t mode);`
`name` est le nom du fichier, `mode` les droits d'accès.

Commande : `mkfifo path`
`path` est le nom du fichier.

Un exemple en ligne de commande

Dans un terminal, on se place dans un répertoire vide, et on entre les commandes suivantes :

```
$ mkfifo test
$ ls -l
  -rw-rw-r-- 1 nlouvet nlouvet 172 févr. 20 15:13 blabla.txt
  prw-rw-r-- 1 nlouvet nlouvet 0 févr. 20 15:14 test
$ cat blabla.txt > test # reste bloqué
$
```

La commande `cat` reste bloquée tant que l'on a pas fait les manipulations suivantes ; dans un autre terminal, dans le même répertoire :

```
$ cat test
  Everyone knows that debugging is twice as hard as writing a program
  in the first place. So if you're as clever as you can be when you
  write it, how will you ever debug it?
$
```

Conclusion

Nous avons vu deux méthodes pour la communication des processus :

- Les signaux : permettent gérer des événements de façon asynchrone, mais ne transmettent que les numéros des signaux. On peut les utiliser pour (dans une certaine mesure) synchroniser des processus.
- Les pipes : communication unidirectionnelle d'octets en mode FIFO ; permet de transférer des volumes de de données importants.

Il existe d'autres méthodes de communications ou de synchronisation :

- files de messages (`man mq_overview`),
- segment de mémoire partagée (`man shm_overview`),
- les sémaphores (`man sem_overview`).

Toutes ces méthodes sont locales à un système donné ; par la suite, vous verrez également les communications *via* le réseau avec les sockets.