Programmation Système II : communication entre processus

Juliusz Chroboczek

5 février 2016

Comme nous l'avons vu, chaque processus est isolé dans son espace de mémoire virtuelle. Afin de partager les données entre les processus, il faut que ceux-ci communiquent.

Dans ce cours, nous verrons quatre techniques de communication entre processus :

- les fichiers ;
- les tubes anonymes *(pipes)* ;
- les tubes nommés ;
- les sockets de domaine Unix.

1 Communication à travers les fichiers

La façon la plus habituelle de transférer les données est d'utiliser un fichier. Un processus *A* écrit les données dans un fichier, que le processus *B* lit ensuite ; le point de rendez-vous entre les deux processus est le nom de fichier. Par exemple, lorsque vous éditez un fichier `hello.c` puis le compilez, la communication entre l'éditeur de texte et le compilateur se fait à travers le fichier `hello.c`.

Lorsque plusieurs processus accèdent au même fichier, il est important que leurs actions soient *synchronisées*.

1.1 Synchronisation de la fin de l'écriture

Si un processus *A* écrit un fichier qu'un processus *B* doit lire, il est important que *B* attende que *A* ait fini avant de lire le fichier. Le cas le plus simple est celui où *A* est le fils de *B*, et *A* termine après l'écriture du fichier — dans ce cas, il suffit que *B* attende la mort de *A* à l'aide de l'appel système `wait` ou `waitpid`.

Dans des cas plus compliqués, *B* ne termine pas immédiatement, ou alors *A* et *B* ne sont pas apparentés ; dans ce cas, il faut utiliser des primitives de communication plus compliquées, par exemple un octet écrit sur un tube nommé ou un signal.

1.2 Cohérence des accès

Lorsque plusieurs processus accèdent au même fichier, il faut qu'ils se coordonnent pour faire des accès cohérents. Il existe deux cas typiques.

Conflit écriture-écriture : *lost update* Si les processus *A* et *B* font simultanément un cycle lecture-modification-écriture sur un fichier *F*, une mise à jour peut être écrasée par l'autre. Considérons par exemple le cas où *A* et *B* listent un entier contenu dans le fichier, lui ajoutent 10, et écrivent le résultat. Si les deux accès ont lieu à peu près au même moment, il se peut que *A* et *B* listent tous deux une valeur *n*, lui ajoutent 10, et écrivent tous deux $n + 10$. La première mise à jour a été perdue — on parle de *lost update* (figure 1).

A	*B*
`read(fd, &x, sizeof(x));`	
	`read(fd, &x, sizeof(x));`
`y = x + 10;`	
	`y = x + 10;`
`write(fd, &y, sizeof(y));`	
	`write(fd, &y, sizeof(y));`

Figure 1 — *Lost update*

Pour éviter ce problème, il faut garantir que chacun des cycles lecture-modification-écriture se fasse de façon atomique, c'est à dire sans pouvoir être interrompu par l'autre.

Conflit écriture-lecture : état incohérent Lorsqu'une mise à jour consiste de plusieurs écritures, il se peut qu'un lecteur observe un état incohérent – un état qui ne doit pas exister. De même que ci dessus, la solution consiste à garantir que toutes les écritures qui constituent une mise à jour se fassent de façon atomique.

1.3 *Locks*

Il existe plusieurs techniques pour garantir l'atomicité relative des opérations effectuées par des processus qui coopèrent. La plus élémentaire consiste à créer atomiquement (`O_EXCL`) un fichier *(lockfile)* au début de la transaction, et à le supprimer à la fin. Le processus qui n'arrive pas à créer le *lockfile* n'a pas le droit de modifier les données partagées. Cette technique a le défaut de ne pas permettre un blocage efficace du processus en attente, et de créer des *lockfile* persisants, qui survivent à la mort de leur créateur.

Unix a une notion de *lock* qui n'a pas ce défaut. En fait, deux.

`flock` L'appel système `flock` permet d'acquérir ou de relâcher de façon atomique un *lock* associé à un fichier donné. Il manipule deux types de *locks* : les *locks exclusifs*, utilisés par un processus qui désire modifier le fichier, et les *locks partagés*, utilisés par un processus qui désire seulement le lire. Les règles sont les suivantes :

– si un processus *A* détient un *lock* exclusif associé à un fichier *F*, aucun autre processus ne peut acquérir de *lock* associé au même fichier, exclusif ou partagé ;
– si un processus *A* détient un *lock* partagé associé à un fichier *F*, d'autres processus peuvent acquérir des *locks* partagés, mais pas des *locks* exclusifs.

L'appel système `flock` a le prototype suivant :

```
int flock(int fd, int operation)
```

Le paramètre `operation` indique l'opération a effectuer ; il vaut `LOCK_SH` ou `LOCK_EX` pour acquérir un *lock*, et `LOCK_UN` pour relâcher un *lock*. Si le *lock* demandé ne peut pas être acquis, l'appel `flock` bloque.

`fcntl` L'appel système `flock`, dû à Unix BSD, est simple, correct et utile ; la norme POSIX d'origine a donc formalisé un appel système différent, `fcntl`, dû à Système V, qui est compliqué, incorrect, et a des fonctionnalités jamais utilisées en pratique. La principale fonctionnalité supplémentaire de `fcntl` est la possibilité d'attacher un *lock* a une partie d'un fichier.

Les locks `fcntl` sont attachés à l'i-nœud mémoire, ce qui les rend inutilisables en pratique ; nous verrons pourquoi c'est le cas dans la partie sur le comportement des descripteurs de fichiers lors d'un `fork`.

Les *locks* sont consultatifs Il est important de souligner que les *locks* obtenus à l'aide de `flock` ou `fcntl` n'intéragissent qu'avec les autres *locks*, et pas avec les appels système `open` ou `write` : rien n'empêche un processus de modifier un fichier protégé par un *lock* — un *lock* ne verrouille rien du tout[1]. Ils ne sont donc utiles que pour garantir l'atomicité relative des opérations effectuées par des processus qui coopèrent.

2 Les tubes

La communication à travers les fichiers a deux défauts : elle crée des structures de données persistantes (qui survivent à la mort des processus), et elle demande une synchronisation externe.

Ce dernier problème est le plus grave. Les *tubes (pipes)* sont une structure de données anonyme qui implémente une file bloquante entre deux processus apparentés. Ils sont décrits en détail dans les notes de L3.

Il est important de noter la sémantique des tubes lorsqu'un des pairs ferme la connexion. Si l'écrivain ferme le tube , le lecteur reçoit une indication de fin de fichier (`read` retourne 0). Si par contre le lecteur ferme le tube le premier, l'écrivain est tué par un signal `SIGPIPE`. La solution consiste à ignorer ce signal (ce qui est décrit dans la suite de ce cours), et alors l'appel à `write` retourne `EPIPE`.

1. Ce pourquoi je n'utilise pas le terme de *verrou* dans ce cours.