Programmation Système III
Partage de descripteurs

Juliusz Chroboczek

22 février 2016

1 Accès multiples aux tubes

Nous avons vu précédemment comment un tube implémente en une seule opération la communication et la synchronisation entre deux processus apparentés, l'écrivain et le lecteur. Mais qu'en est-il lorsqu'un tube est partagé par plus de deux processus ?

Rappelons tout d'abord qu'un tube implémente une communication par *flots d'octets* : les frontières des `read` et `write` ne sont pas préservées, un seul `write` peut être scindé du côté du lecteur en plusieurs `read` et, inversément, plusieurs `write` peuvent être coalescés. Les lectures et écritures partielles sont possibles lorsque le tampon est presque vide ou presque plein, et il ne suffit donc pas d'utiliser un tampon simple pour accéder à un tube — il faut maintenir un pointeur de début de données ou alors utiliser un tampon circulaire (voir notes de cours de L3).

Atomicité des écritures Lorsque plusieurs écrivains partagent un même tube, les données d'un seul `write` se retrouvent contigües[1]. Cependant, les écritures partielles peuvent faire que les données de plusieurs écrivains se trouvent entrelacées. Pour compenser ce problème, une écriture inférieure à `PIPE_MAX` octets est toujours atomique : soit elle bloque, soit elle réussit en entier, et POSIX garantit que `PIPE_MAX` vaut au moins 512. (Sous Linux, `PIPE_MAX` vaut 4096, mais vous ne devriez pas compter dessus.)

Atomicité des lectures Les lectures depuis les tubes sont en principe atomiques — s'il n'est pas possible de satisfaire une lecture avec des données contigües, le système effectue une lecture partielle.

2 Tubes nommés

Les tubes « anonymes » ordinaires ne permettent de communiquer qu'entre processus apparentés : un tube qui permet aux processus *A* et *B* de communiquer doit être créé dans un ancêtre

1. Mais on me dit que Mac OS X est boggué.

commun de *A* et *B*. Un *tube nommé* est un tube qui vit dans le système de fichiers — il a donc un nom qui sert de point de rendez-vous aux processus qui désirent communiquer.

Un tube nommé est créé à l'aide de la fonction `mkfifo` (qui, sous le capot, appelle l'appel système `mknod`). Les processus y accèdent en utilisant l'appel système `open`, comme pour un fichier ordinaire.

Les tubes nommés synchronisent les lectures et les écritures comme les tubes ordinaires. Pour ce qui est des ouvertures, la sémantique est la suivante :

- une ouverture en écriture bloque jusqu'à ce qu'il y ait un lecteur ;
- une ouvertire en lecture bloque jusqu'à ce qu'il y ait un écrivain.

À la différence des *sockets*, les tubes nommés ne font pas de démultiplexage (il n'y a pas de `accept`) : les données des différents écrivains apparaissent comme un seul flot. De ce fait, leur utilisation avec plusieurs écrivains demande d'utiliser un protocole de niveau supérieur ; en pratique, ils sont rarement utilisés, sauf par les serveurs de *logs*.

3 *Sockets* de domaine Unix

Un *socket* de domaine Unix est un type de *socket* qui vit dans le système de fichiers et ne permet que la communication locale à l'hôte. Ils ont une sémantique semblable aux *sockets* de domaine IP ou IPv6 que vous connaissez, et il en existe deux variantes, `SOCK_STREAM` (communication fiable par flots) et `SOCK_DGRAM` (communication non-fiable par datagrammes).

Les *sockets* de domaine Unix sont créées par un appel à `socket` avec le premier paramètre valant `PF_UNIX`. Leurs adresses sont représentées par une structure `sockaddr_un` :

```
struct sockaddr_un {
    sa_family_t sun_family;
    char sun_path[108];
};
```

Le champ `sun_family` vaut `AF_UNIX`. Le champ `sun_path` est le nom de la *socket* (un « chemin » dans le système de fichiers).

Les *sockets* Unix ont deux autre fonctionnalités, la possibilité d'annoncer son *pid* de façon sécurisée et de transférer un descripteur de fichier ouvert à un autre processus. Nous les verrons peut-être en TP.

4 Duplication de fichiers ouverts — `fork` et `dup`

Losqu'un descripteur de fichier est créé, par exemple à l'aide de `open`, `pipe` ou `socket`, le système créé une structure de données appelée un *i-nœud mémoire* (figure 1). Le système crée ensuite une deuxième structure de données, l'*entrée de la table de fichiers ouverts*, et un descripteur de fichier local au processus qui réfère à cette entrée.

Lorsqu'un même fichier est ouvert deux fois (`open` suivi de `open`), deux entrées distinctes sont créées dans la table de fichiers ouverts. Par contre, si un fichier ouvert est dupliqué (à l'aide de `fork`, `dup` ou `dup2`), les deux descripteurs de fichiers réfèrent à la même entrée, qui n'est donc pas

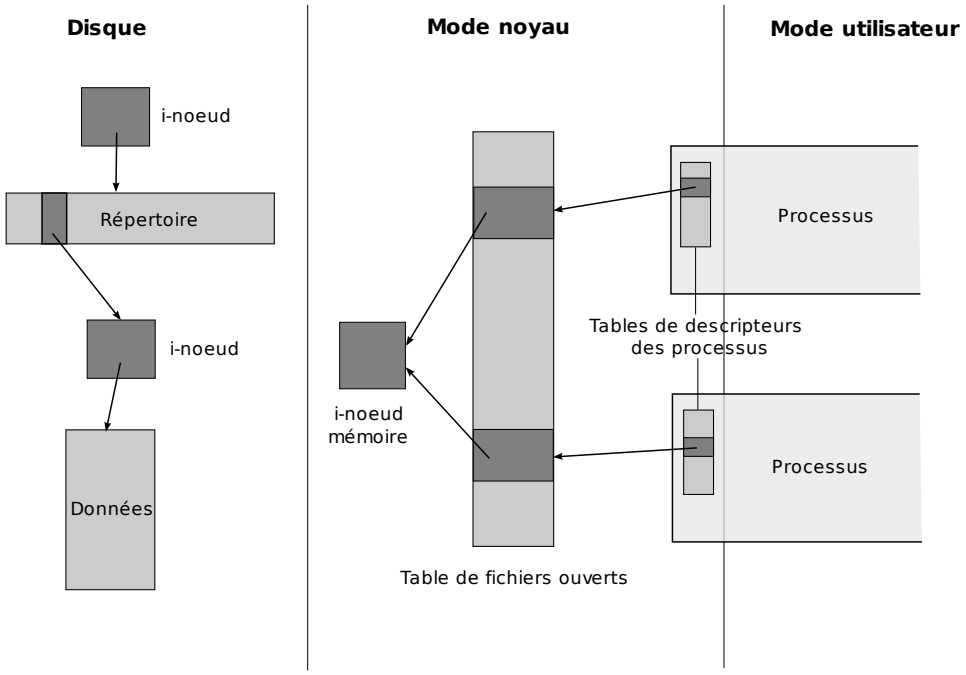

Figure 1 — Structures de données sur disque et en mémoire

dupliquée. La distinction est importante lorsque des structures de données mutables se trouvent dans l'entrée de fichier ouvert.

Pointeur de position courante Le *pointeur de position courante* est un entier qui indique à quelle position se fera la prochaine écriture. Il est consulté et mis à jour par chaque `read` et `write`, et manipulé explicitement à l'aide de `lseek`.

Le pointeur de position courante se trouve dans l'entrée de la table de fichiers ouverts. De ce fait, un descripteur de fichier dupliqué à l'aide de `fork` ou `dup2` a le même pointeur de position courante que le descripteur d'origine. Le fragment de code suivant écrit `totototo` :

```
int fd = open(...);
if(fd < 0) ...;
pid = fork();
rc = write(fd, "toto", 4);
```

`flock` Un *lock* obtenu à l'aide de `flock` est rattaché à l'entrée de la table de fichiers ouverts. De ce fait, après la séquence `flock`, `fork`, les deux processus détiennent le *lock*. A contrario, lorsqu'un processus ouvre le même fichier deux fois, il ne peut détenir un *lock* exclusif que sur un des descripteurs de fichier obtenus. Un *lock* est perdu lorsque le dernier descripteur de fichier référant à une entrée de table de fichiers ouverts est fermé.

Cette sémantique permet, si l'on est soigneux, d'écrire des programmes fiables à l'aide de `flock`.

`fcntl` Un *lock* obtenu à l'aide de `fcntl` est rattaché au processus. Il n'est donc pas dupliqué par `fork` ou `dup`, et il est perdu dès lors que le processus effectue `close` sur le fichier donné. Par exemple, la séquence `dup-close` cause la perte d'un *lock*, même si elle est effectuée par une bibliothèque que le programmeur ne contrôle pas.

Cette sémantique empêche d'écrire des programmes fiables à l'aide de `fcntl`, même si on est soigneux.

Open file description locks Les versions récentes de Linux implémentent les *open file description locks*, une forme de *lock* non-standard qui combinent la sémantique de `flock` avec la complexité de `fcntl`. Ils seront probablement inclus dans la prochaine édition de POSIX (2018 ?).