Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Cité
L3 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2023-2024

QCM n° 2

vendredi 15 décembre 14h15-15h15

Amphis 2A et 8C

Communication par tubes

Un tube, qu'est-ce que c'est ?

- un mécanisme de communication entre processus,
- manipulable avec descripteur, `read`, `write`...
- lecture destructrice : tout octet lu est consommé et retiré du tube,
- flot continu : pas de séparation entre 2 écritures successives,
- fonctionnement de type fifo, unidirectionnel : un tube a une extrémité en écriture et une en lecture,
- capacité limitée (donc notion de tube plein)
- par défaut, les opérations sur les tubes sont ***bloquantes***

auto-synchronisant : impossible de lire un caractère avant qu'il ne soit écrit !

importance de deux nombres :

- nombre de lecteurs (= nombre de *descripteurs* en lecture) : s'il est nul, toute écriture est interdite (`SIGPIPE`)
- nombre d'écrivains : le comportement en lecture sur un tube vide dépend de sa nullité.

Création d'un tube (anonyme)

```
int pipe(int pipefd[2]);
```

- crée et ouvre un tube anonyme – donc alloue :
  - un i-nœud mémoire,
  - 2 entrées dans la table des ouvertures de fichiers (une en lecture, une en écriture),
  - 2 descripteurs pour ces 2 ouvertures,
- stocke ces descripteurs dans `pipefd` : lecture dans `pipefd[0]`, écriture dans `pipefd[1]`,
- renvoie 0 en cas de succès, -1 en cas d'échec (si la table de descripteurs du processus ou la table des ouvertures de fichiers est pleine)

le tube créé n'est accessible que via ces 2 descripteurs – comme il n'a pas de nom, on ne peut pas le réouvrir avec `open`.
$\Longrightarrow$ seuls les descendants du processus qui a créé un tube anonyme peuvent donc y accéder, en héritant des descripteurs.

Lecture dans un tube

```
char buf[TAILLE_BUF]; int tube[2]; pipe(tube);
...
ssize_t nb_lus = read(tube[0], buf, TAILLE_BUF);
```

- *si le tube n'est pas vide* et contient `taille` octets, `nb_lus = min(taille, TAILLE_BUF)` octets sont extraits et copiés dans `buf`,
- *si le tube est vide*, le comportement dépend du nombre d'écrivains (*i.e.* de descripteurs en écriture sur le tube) :
  - renvoie `nb_lus = 0` si le nombre d'écrivains est nul,
  - sinon, par défaut la lecture est ***bloquante*** : le processus est mis en sommeil jusqu'à ce que quelque chose change (contenu du tube ou nombre d'écrivains)

le caractère bloquant permet la synchronisation d'un lecteur sur un écrivain... mais peut également provoquer des ***auto- ou interblocages***!

Situations de blocages typiques

à un seul processus (autoblocage)

```
int tube[2]; pipe(tube);
...
/* tube vide et le processus est le seul écrivain */
read(tube[0], buf, 1);
```

à deux processus (interblocage)

```
int tube1[2], tube2[2]; pipe(tube1); pipe(tube2);
if (fork() == 0) {
  read(tube1[0], buf1, 1); // blocage sur tube1 vide
  write(tube2[1], buf2, 1);
}
else {
  read(tube2[0], buf1, 1); // blocage sur tube2 vide
  write(tube1[1], buf2, 1);
}
```

Règle d'or : *toujours* libérer les descripteurs inutiles

Écriture dans un tube

```
char buf[TAILLE_BUF]; int taille; int tube[2]; pipe(tube);
...
ssize_t nb_ecrits = write(tube[1], buf, taille);
```

- *si le nombre de lecteurs est nul*, le signal `SIGPIPE` est délivré, ce qui provoque par défaut la terminaison du processus ; si le signal est ignoré, retour -1 et `errno=EPIPE`
- *sinon*, par défaut l'écriture est ***bloquante*** : si l'écriture n'est pas réalisable (en particulier si le tube est plein), le processus est mis en sommeil jusqu'à ce que la situation change.

garantie d'atomicité : si `taille` <= `PIPE_BUF`, les caractères sont écrits en une fois (donc l'écriture bloque tant que ce n'est pas possible)
aucune garantie si `taille` > `PIPE_BUF` $\Longrightarrow$ à éviter absolument

`PIPE_BUF` vaut au moins 512 (4096 sous Linux)

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

Utilisation pour les redirections (*pipelines*)

`cmd1 | cmd2 | cmd3 | ... | cmdn`

- n processus pour les n commandes
- $n-1$ tubes pour les relier
- le processus exécutant `cmd`i lit dans le tube $(i-1)$ (si $i > 1$) et écrit dans le tube i (si $i < n$)

Utilisation pour les redirections (*pipelines*)

`cmd1 | cmd2 | cmd3 | ... | cmdn`

- n processus pour les n commandes
- $n-1$ tubes pour les relier
- le processus exécutant `cmd`i lit dans le tube $(i-1)$ (si $i > 1$) et écrit dans le tube i (si $i < n$)

le tube i doit donc être créé *avant* le `fork` qui sépare les processus exécutant `cmd`i et `cmd`$(i+1)$, ce qui laisse beaucoup de possibilités de généalogie (lignée dans un sens ou l'autre, père supervisant n fils...)

Attention à ***toujours*** fermer les descripteurs inutilisés !!

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

- pour que la terminaison d'un processus entraîne celle de tous les processus, il faut *impérativement* que les descripteurs inutilisés soient fermés

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

- pour que la terminaison d'un processus entraîne celle de tous les processus, il faut *impérativement* que les descripteurs inutilisés soient fermés
- pour que le shell reprenne la main à l'issue de l'exécution (et pas avant, en particulier après l'affichage de la sortie de `cmdn`), il faut que le dernier processus qui termine soit son fils

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

- pour que la terminaison d'un processus entraîne celle de tous les processus, il faut *impérativement* que les descripteurs inutilisés soient fermés
- pour que le shell reprenne la main à l'issue de l'exécution (et pas avant, en particulier après l'affichage de la sortie de `cmdn`), il faut que le dernier processus qui termine soit son fils

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

- pour que la terminaison d'un processus entraîne celle de tous les processus, il faut *impérativement* que les descripteurs inutilisés soient fermés
- pour que le shell reprenne la main à l'issue de l'exécution (et pas avant, en particulier après l'affichage de la sortie de `cmdn`), il faut que le dernier processus qui termine soit son fils

- une généalogie linéaire `cmdn -> ... -> cmd2 -> cmd1` est préférable à `cmd1 -> cmd2 -> ... -> cmdn`

Utilisation pour les redirections (*pipelines*)

```
cmd1 | cmd2 | cmd3 | ... | cmdn
```

- pour que la terminaison d'un processus entraîne celle de tous les processus, il faut *impérativement* que les descripteurs inutilisés soient fermés
- pour que le shell reprenne la main à l'issue de l'exécution (et pas avant, en particulier après l'affichage de la sortie de `cmdn`), il faut que le dernier processus qui termine soit son fils

- une généalogie linéaire `cmdn -> ... -> cmd2 -> cmd1` est préférable à `cmd1 -> cmd2 -> ... -> cmdn`
- une généalogie avec n fils « supervisés » par leur père est encore mieux

Tubes nommés

de même nature que les tubes anonymes, mais avec une existence dans le SGF :

- création et ouverture séparées
- accessibles par des processus non nécessairement apparentés
- accessibilité contrôlable
- persistants (enfin... pas leur contenu)

Création

```
int mkfifo(const char *pathname, mode_t mode);
```

- les paramètres ont la même signification que pour `creat()`, `mkdir()` ou `open()`
- renvoie -1 en cas d'erreur, 0 sinon

Création

```
int mkfifo(const char *pathname, mode_t mode);
```

- les paramètres ont la même signification que pour `creat()`, `mkdir()` ou `open()`
- renvoie -1 en cas d'erreur, 0 sinon

Suppression avec `unlink()`, renommage avec `rename()`, changement des droits avec `chmod()`... comme les autres entrées de répertoires

Tubes nommés

Ouverture avec `open()`, mais avec une sémantique particulière : par défaut, les ouvertures de tubes sont ***bloquantes***, c'est-à-dire :

- une ouverture en lecture bloque si (*i.e.* tant qu')il n'y a pas d'écrivain ;
- une ouverture en écriture bloque si (*i.e.* tant qu')il n'y a pas de lecteur.

$\Longrightarrow$ point de rendez-vous entre deux processus

une ouverture renvoie ***un*** descripteur, donc sur un seul bout du tube : il faut donc choisir entre `O_RDONLY` et `O_WRONLY`

sous Linux (mais non normalisé POSIX), une ouverture en `O_RDWR` est possible, et ne bloque pas

Pourquoi utiliser des tubes nommés ?

pour dupliquer le flot de sortie : chaque processus d'un pipeline `cmd1 | cmd2 | ... | cmdn` consomme les données qu'il reçoit ; comment envoyer la sortie de `cmd1` sur `cmd2` *et* `cmd3` sans utiliser de fichiers ordinaires intermédiaires ?

Exemple : lister tous les fichiers du répertoire courant et, à la fois :

- les compter
- afficher les 3 plus gros

```
$ mkfifo /tmp/tube;
$ wc -l < /tmp/tube &
$ ls -l | tee /tmp/tube | sort -k5n | tail -3
```

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes — atomicité, *cf* `PIPE_BUF`
- délimitation des requêtes

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes — atomicité, *cf* `PIPE_BUF`
- délimitation des requêtes — ... pas de garantie liée au tube

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes — atomicité, *cf* `PIPE_BUF`
- délimitation des requêtes — ... pas de garantie liée au tube

solutions :

- utiliser un marqueur de fin

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes — atomicité, *cf* `PIPE_BUF`
- délimitation des requêtes — ... pas de garantie liée au tube

solutions :

- utiliser un marqueur de fin
- imposer des requêtes de taille fixe

Pourquoi utiliser des tubes nommés ?

pour concevoir une architecture client-serveur :

- les parties conviennent d'une référence pour un tube de requêtes
- le tube est soit persistant, soit créé au démarrage par le serveur
- les clients écrivent leurs requêtes sur le tube
- le serveur lit les requêtes sur le tube et les traite

problèmes potentiels :

- entrelacement des requêtes — atomicité, *cf* `PIPE_BUF`
- délimitation des requêtes — ... pas de garantie liée au tube

solutions :

- utiliser un marqueur de fin
- imposer des requêtes de taille fixe
- préfixer chaque message d'un entête de taille fixe

Et si la requête appelle une réponse ?

impossible de l'envoyer via un tube partagé par plusieurs clients ! (et encore moins via le tube de requêtes...)

il faut un tube de réponse par client – donc il faut :

- avoir une convention de nommage (et de création),
- prendre garde à ne pas provoquer d'interblocage à l'ouverture

Mode non bloquant

Parfois le comportement bloquant par défaut des tubes n'est pas adapté ; on peut le modifier :

- à l'ouverture d'un tube nommé, avec le flag `O_NONBLOCK`
- (sous Linux) à la création/ouverture d'un tube anonyme par `int pipe2(int pipefd[2], int flags)`, avec le flag `O_NONBLOCK`
- après l'ouverture, en modifiant les flags du descripteur avec `int fcntl(int fd, int cmd, ... /*arg */)`

comportement d'une ouverture non bloquante :

- en lecture, elle réussit immédiatement ;
- en écriture, elle réussit seulement en présence d'un lecteur ; sinon elle échoue, avec `errno=ENXIO`

puis les lectures ou écritures seront non bloquantes

Mode non bloquant

comportement des lectures non bloquantes : comme les lectures bloquantes ***sauf*** si le tube est vide et le nombre d'écrivains non nul : retourne `-1` immédiatement, avec `errno=EAGAIN`

comportement des écritures non bloquantes : comme les écritures bloquantes ***sauf*** si le nombre de lecteurs est non nul et le tube plein (*i.e.* trop rempli pour réaliser l'écriture) :

- si `taille <= PIPE_BUF`, pour respecter à la fois le caractère non bloquant et la garantie d'atomicité, retourne `-1` immédiatement, sans écriture, avec `errno=EAGAIN`
- si `taille > PIPE_BUF`, réalise une ***écriture partielle*** et retourne immédiatement le nombre de caractères écrits (ou `-1` avec `errno=EAGAIN`

Bascule bloquant — non bloquant

la fonction `fcntl` est un outil multi-usage de contrôle des fichiers ouverts ; en particulier :

```
int fcntl(int fd, F_GETFL); /* retourne les flags de l'ouverture */
int fcntl(int fd, F_SETFL, int value);/* définit de nouveaux flags */
```

Bascule bloquant — non bloquant

la fonction `fcntl` est un outil multi-usage de contrôle des fichiers ouverts ; en particulier :

```
int fcntl(int fd, F_GETFL); /* retourne les flags de l'ouverture */
int fcntl(int fd, F_SETFL, int value);/* définit de nouveaux flags */
```

pour basculer en mode non bloquant, il faut *ajouter* le flag `O_NONBLOCK` – c'est un « OU » bit-à-bit – alors que pour basculer en mode bloquant, il faut remettre le bit à 0 – c'est donc un « ET » avec la négation de `O_NONBLOCK`

```
int attributs = fcntl(fd, F_GETFL);
if (attributs == -1) ...
/* passage en mode non bloquant : */
fcntl(int fd, F_SETFL, attributs | O_NONBLOCK);
/* passage en mode bloquant : */
fcntl(int fd, F_SETFL, attributs & ~O_NONBLOCK);
```