

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 2022-2023

QCM en début d'amphi le 23 novembre

COMMUNICATION PAR TUBES

UN TUBE, QU'EST-CE QUE C'EST ?

- un mécanisme de communication entre processus,
- manipulable presque comme un fichier ordinaire – descripteur, `read`, `write`...
- la lecture est `destructrice` : tout octet lu est consommé et retiré du tube,
- `flot continu` de caractères : 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*

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

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`.

⇒ 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`

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.

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.

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` \leq `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` \implies 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 `cmdi` 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 `cmdi` 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)

TUBES NOMMÉS

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

TUBES NOMMÉS

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.

⇒ 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