Module SY5 – Systèmes d'Exploitation

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université de Paris (Diderot)
L3 Informatique & DL Bio-Info, Jap-Info, Math-Info
Année universitaire 2021-2022

Communication par tubes (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
- 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...)

plus généralement, un tube avec plusieurs lecteurs potentiels est très difficile à gérer : il faut être sûr que chaque message sera lu en une seule fois, et que le destinataire sera le bon... c'est essentiellement impossible, sauf si

- les messages ont une taille fixe
- les lecteurs sont interchangeables (typiquement, un serveur distribué)

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

Et si un processus doit surveiller plusieurs descripteurs ?

que faire quand un processus doit effectuer, sans ordre prédéfini, plusieurs opérations susceptibles de bloquer ?

exemples de telles situations :

- un serveur qui reçoit des requêtes de clients différents sur des tubes différents
- un client qui doit à la fois lire les demandes de l'utilisateur sur son entrée standard + les réponses du serveur sur un tube
- non limité à la lecture : par exemple, un serveur peut avoir des requêtes à lire et des réponses à donner, mais dans un/des tube(s) plein(s)

Et si un processus doit surveiller plusieurs descripteurs ?

que faire quand un processus doit effectuer, sans ordre prédéfini, plusieurs opérations susceptibles de bloquer ?

- solution 1 : cloner le processus pour distribuer les tâches
  problème : ce n'est pas toujours possible, et cela pose des problèmes de synchronisation
- solution 2 : passer les descripteurs concernés en mode non bloquant, et les consulter en boucle
  problème : attente active, très consommatrice de ressources
- solution 3 : faire appel au système d'exploitation pour qu'il surveille les descripteurs à notre place

Scrutation de descripteurs

principe : le processus définit des ensembles de descripteurs à surveiller (en lecture ou en écriture), puis se met en attente (bloquante) jusqu'à ce qu'un d'entre eux soit prêt.

possibilité de définir un temps d'attente maximal

deux primitives existent (héritage de BSD et System V) :

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                fd_set *exceptfds, struct timeval *timeout);

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Scrutation de descripteurs : select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

- `ndfs` est strictement supérieur au plus grand descripteur à surveiller,
- `readfds` est un pointeur sur un ensemble de descripteurs en lecture
- `writefds` est un pointeur sur un ensemble de descripteurs en écriture
- `timeout` est le temps d'attente maximal ou `NULL` pour un temps infini.
  (deux champs, `tv_sec` et `tv_usec`, en secondes et microsecondes),

`exceptfds` sera toujours `NULL` ; `readfds` et `writefds` éventuellement aussi.

les ensembles sont initialisés avec :

```
void FD_ZERO(fd_set *set);      /* initialise un ensemble vide */
void FD_SET(int fd, fd_set *set);   /* ajoute fd à set */
void FD_CLR(int fd, fd_set *set);   /* enlève fd de set */
```

Scrutation de descripteurs : select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

l'appel bloque jusqu'à ce que l'une des conditions suivantes soit vérifiée :

- un des descripteurs de `readfds` ou de `writefds` est « prêt »,
- le temps imparti est écoulé,
- le processus a reçu un signal.

« prêt » signifie qu'une tentative de lecture (ou d'écriture) ne bloquera pas, éventuellement parce qu'elle échouera immédiatement.

`select` retourne le nombre de descripteurs prêts, éventuellement 0 si le temps imparti est échu (et -1 en cas d'erreur)

les ensembles `readfds` et `writefds` sont modifiés et décrivent les ensembles de descripteurs prêts, testables par :

```
int FD_ISSET(int fd, fd_set *set); /* teste si fd est dans set */
```

Scrutation de descripteurs : select

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
                  fd_set *exceptfds, struct timeval *timeout);
```

les ensembles `readfds` et `writefds` sont modifiés et décrivent les ensembles de descripteurs prêts

Corollaire : pour des appels répétés à `select`, il faut rétablir les ensembles à surveiller avant chaque appel !

autre joyeuseté : sous Linux, `timeout` est modifié pour indiquer le temps non écoulé ; mais ce n'est pas normalisé POSIX...

Scrutation de descripteurs : poll

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

se comporte de manière analogue, mais les paramètres sont passés de manière différente :

- le temps imparti `timeout` est simplement exprimé en millisecondes (0 pour ne pas bloquer, -1 pour bloquer indéfiniment)
- `fds` pointe sur un tableau de structures décrivant les descripteurs à surveiller :

```
struct pollfd {
  int   fd;         /* descripteur; ignoré si négatif */
  short events;     /* événements à surveiller : POLLIN, POLLOUT*/
  short revents;    /* événements observés : POLLIN, POLLOUT, POLLHUP,
POLLNVAL... */
};
```

corollaire, `poll` ne détruit pas ses paramètres, qui sont donc réutilisables