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 2022-2023

Processus (encore)

Rappel : création de processus

```
pid_t fork(void);
```

- retourne -1 en cas d'erreur (et `errno` est positionnée)

sinon :

- crée un nouveau processus (*fils*) par clonage du processus courant (*père*)
- retourne 0 dans le processus fils
- retourne le pid du fils dans le processus père

autrement dit, ***un appel*** à cette fonction entraîne ***deux retours***

le nouveau processus ne diffère de l'ancien essentiellement que par son identifiant (et celui de son père)

le fils poursuit l'exécution au point où en était son père

Création et héritage

```
pid_t fork(void);
```

l'*espace d'adressage* du fils est une *copie* de celui du père

conséquence : toutes les variables sont *initialisées aux mêmes valeurs* que celles du père – mais ce ne sont *pas les mêmes variables*, elles évoluent ensuite indépendamment

attention, cela s'applique aussi aux variables « cachées », comme les tampons de la bibliothèque standard par exemple

la *table des descripteurs* du fils est (initialement) une copie de celle du père : chaque descripteur du fils pointe sur la *même entrée* de la table des ouvertures de fichiers que celui du père

ils partagent donc la *même position courante* dans le fichier ouvert

Comment faire communiquer les processus ?

chaque processus est *isolé* dans son espace mémoire

donc deux processus, même s'ils sont père et fils, ne peuvent pas partager des données simplement par l'intermédiaire des variables situées dans leur espace mémoire (spécifique à chaque processus)

solution (très) partielle : utiliser la valeur de retour du fils – donc seulement pour une valeur entre 0 et 255, à transmettre d'un fils vers son père, et seulement à la terminaison du fils

pour tous les autres cas, il faut utiliser un stockage *extérieur* – par exemple un fichier

... ce qui pose un problème de *synchronisation* des processus, plus général que la synchronisation d'un père sur la terminaison d'un fils

donc il faut d'autres mécanismes de synchronisation que `wait`

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`.
$\Longrightarrow$ 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