

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

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 (*fil*s) par clonage du processus courant (*p*ère)
- 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

POURQUOI SÉPARER CLONAGE ET RECouvreMENT ?

principalement car cela laisse une opportunité pour modifier certaines choses – impérativement parmi celles qui ne seront pas écrasées par le recouvrement

POURQUOI SÉPARER CLONAGE ET RECouvreMENT ?

principalement car cela laisse une opportunité pour modifier certaines choses – impérativement parmi celles qui ne seront pas écrasées par le recouvrement

cela concerne principalement :

- la table des descripteurs : changer les fichiers (avec `dup()` et `dup2()`) associés aux descripteurs « standard » (`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO` permet de réaliser des redirections
- la définition des comportements associés à la réception de signaux (cf. cours ultérieur)

DUPLICATION DE DESCRIPTEURS

dupliquer un descripteur, c'est associer à un nouveau descripteur la même ouverture de fichier (*i.e.*, la même entrée de la table des ouvertures) – exactement comme ce qui se passe lors d'un **fork** : tous les descripteurs du processus père sont dupliqués pour définir les descripteurs du processus fils (*et le compteur de descripteur associé à l'ouverture est incrémenté*)

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

- duplique le descripteur **oldfd**,
- retourne la valeur du nouveau descripteur

la valeur du nouveau descripteur est :

- le plus petit entier disponible, dans le cas de **dup**,
- **newfd**, dans le cas de **dup2**; le cas échéant, **newfd** est d'abord fermé (*opération atomique*)

COMMENT FAIRE COMMUNIQUER LES PROCESSUS ?

chaque processus est *isolé* dans son espace mémoire

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)

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

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

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

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,

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

- un mécanisme de communication entre processus,
- manipulable presque comme un fichier ordinaire – descripteur, `read`, `write`...

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,

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,

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,

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`)

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, 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, 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)

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`,

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) :

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,

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)

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)

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);
```

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);  
}
```

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)