Sytèmes 1 — Processus

December 14, 2011

Contents

1 Le concept de processus

Un processus – l'exécution d'un programme binaire par la machine.

L'espace d'adressage: instructions, données (dont la pile d'exécution et le tas). Cette espace est dynamique et change avec le temps.

L'exécution de deux modes différents:

- **mode utilisateur** (user mode) – le processus exécute ses propres instructions et accède les données de son espace d'adressage. La tentative d'accéder à des données hors de son espace d'adressage provoque un échec et envoie d'un signal.
- Dans le **mode système, noyau, superviseur** (supervisor, system, kernel mode) le processus exécute des instruction n'appartenant pas au programme mais au noyau, cela permet d'accéder à l'ensemble de données de système (tables de système). Changement de mode via l'appel de système par une trappe (trap) ou par un évènement (interruption). Dans le dernier cas le processus exécute le gérant de l'interruption (handler) correspondant et reprend au retour de handler.

L'ordonnancement (schedulling) de processus. L'utilisation de priorités. Les caractéristiques de processus dans le bloc de contrôle (BCP).

BCP composé de deux parties:

1. une partie conservée dans le noyau. Contient les information utiles pour le noyau même si le processus inactif: identité, les infos disponible uniquement en mode noyau,
2. les informations dans l'espace d'adressage de processus: descripteurs de fichiers, handlers de signaux.

2 Attributs d'un processus

Chaque processus possède un certain nombre d'attributs. Cette section donne la liste de fonctions qui permettent d'obtenir les différents attributs.

2.1 L'identité

L'identité de processus, **pid**, est un entier positif unique qui permet d'identifier le processus.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void); //l'identite du processus
pid_t getppid(void); //l'identite du processus pere
```

`pid_t` c'est un type entier signé. `getpid()` donne le pid de processus qui ne change pas de la création jusqu'à la mort du processus. `getppid()` donne l'identité du processus père. Cette identité peut changer si le père meure.

2.2 Propriétaire

Chaque processus possède un propriétaire (identifié par une valeur de type `uid_t`, c'est une valeur entière) et un groupe de propriétaires (du type `gid_t`, aussi un type entier).

En fait il y a même deux propriétaires et deux groupes propriétaires, propriétaire réel et le propriétaire effectif et le groupe propriétaire réel et effectif.

On reviendra plus tard sur les différents propriétaires. Pour l'instant juste les fonctions qui permettent de connaître le propriétaire:

Propriétaire réel

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void); /*retourne l'identite du proprietaire reel*/
```

Le propriétaire réel c'est celui qui a lancé le processus. Si vous lancez un processus depuis le terminale c'est le propriétaire de processus l'interprète de commande qui contrôle le terminal.

Propriétaire effectif

```
uid_t geteuid(void); /* retourne id du proprietaire effectif*/
```

C'est le propriétaire effectif qui détermine les droits d'accès aux ressources (fichiers et processus).

```
int setuid(uid_t uid)
```

permet de changer le propriétaire réel et effectif. L'utilisateur non privilégié ne peut modifier que le propriétaire effectif en lui donnant la valeur du propriétaire réel.

Groupe propriétaire réel

```
gid_t getgid(void)
```

Groupe propriétaire effectif

```
gid_t getegid(void)
```

```
int setgid(gid_t)
```

Permet de changer le groupe propriétaire réel et effectif. L'utilisateur non privilégié peut modifier uniquement le groupe propriétaire effectif en lui attribuant la valeur de groupe propriétaire réel.

2.3 Répertoire de travail

Chaque processus possède le répertoire de travail utilisé pour résoudre les références relatives.

Le répertoire de travail est référencé par . (point).

On peut modifier le répertoire de travail avec :

```
#include <unistd.h>
int chdir(const char *reference)
```

où la référence donne le chemin vers le nouveau répertoire de travail. `chdir()` retourne 0 en cas de succès et -1 en cas d'échec.

Pour obtenir la référence absolue (le chemin absolu) de répertoire de travail on utilise la fonction :

```
#inlcude <unistd.h>
char *getcwd(char *tampon, size_t taille)
```

Si l'appel réussi le tampon contient le chemin absolu vers le répertoire de travail. `tampon` doit contenir une adresse valide du tampon où `getcwd()` doit mettre le résultat, `taille` donne la taille du tampon.

La valeur retournée par `getcwd()` est `tampon` si appel réussi et `NULL` dans le cas contraire. Les erreurs possibles: `EACCESS` problème d'accès, `ERANGE` le tampon trop petit. Si le tampon trop petit il faut agrandir et réessayer.

2.4 Session et groupe de processus

Une **session** est composée de processus lancés par utilisateur après la connexion. Les sessions sont utiles pour gérer les déconnexions de l'utilisateur pour provoquer la terminaison des tous les processus lancés par l'utilisateur.

Les **groupes de processus** sert au contrôle de tâches (job control). Permet de détacher les tâches en arrière plan et les ramener en avant plan.

2.4.1 Terminal de contrôle

Un terminal — un fichier spécial permettant d'effectuer les entrées sorties asynchrones. Fonctionne en mode *full duplex* c'est-à-dire traite les entrés et sorties en parallèle (claver/écran).

Le terminal de contrôle est désigné par `/dev/tty`

2.5 Date de création

La date de création d'un processus est donnée en nombre de secondes depuis EPOCH (1 janvier 1970).

2.6 Temps CPU

Pour connaître le temps CPU pour les deux modes: utilisateur et noyau, on utilise la fonction `times` en lui passant l'adresse de la structure `tms`:

```
#include <unistd.h>
clock_t times(struct tms *buf)
```

où `clock_t` un type entier. La fonction met à jour le contenu de la structure `struct tms`. La fonction retourne `(clock_t) -1` en cas d'echec et mets dans `errno` le code d'erreur.

Dans ce qui suit le temps `clock_t` est toujours mesuré en nombre de clics d'horloge. La structure `struct tms` définie dans

```
#include <sys/times.h>
struct tms{
  clock_t  tms_utime;  user CPU time
  clock_t  tms_stime;  system CPU time
  clock_t  tms_cutime; user CPU time of terminated child processes
  clock_t  tms_cstime; system CPU time of terminated child processes
}
```

Le temps `tms_utime` et `tms_stime` d'un fils sont ajoutés dans `tms_cutime` et `tms_cstime` pour chaque fils terminé au moment où `wait()` ou `waitpid()` retourne le *pid* de ce fils.

- `tms_utime` le temps que le processus a passé en mode utilisateur (en exécutant les instructions de l'utilisateur).
- `tms_stime` le temps que le processus a passé en mode système,
- `tms_cutime` — c'est la somme de `tms_utime` de processus courant et de `tms_cutime` de chaque processus fils terminé et attendu avec `wait()` oun `waitpid()`.
- `tms_cstime` — est la somme de `tms_stime` de processus courant et de `tms_stime` de chaque fils terminé et attendus avec `wait()` up `waitpid()`.

Pour déterminer le nombre de clics d'horloge par seconde on utilise `sysconf(_SC_CLK_TCK)`. Ce nombre change d'une machine à l'autre.

2.6.1 Le masque de création de fichiers

Le masque de création de fichier est une propriété de processus. Pour changer de masque:

```
#include <unistd.h>
mode_t umask(mode_t masque)
```

Pour les détails voir les explications dans la partie FICHIER.

2.7 Table de descripteurs

Le processus fils herite la table de descripteurs de processus père.

3 Espace d'adressage de processus

L'espace d'adressage virtuel d'un processus est divisé en plusieurs segments :

- la région de code (text segment) contient les instructions que la machine doit exécutées, souvent ce segment est accessible uniquement en lecture pour que le programme ne modifie par erreur son propre code.
- le segment de données initialisées (initialized data segment) contient les variables déclarées en dehors de fonctions et initialisées explicitement dans le programme. Par exemple si le programme contient la déclaration

  ```
  int toto=99;
  ```

à l'extérieur de toute fonction C alors la mémoire pour la variable `toto` se trouve dans ce segment est initialisée avec la valeur 99 au moment de chargement de programme pour exécution.

- le segment de données non-initialisées contient par les variables de niveau 0 (définies en dehors de fonctions) non-initialisées. Par exemple si

  ```
  long tab[20];
  ```

  est défini en dehors de toute fonction alors la mémoire pour `tab` se trouve dans ce segment et est initialisée à 0 par le noyau au moment de chargement de programme dans la mémoire.

- le tas (heap) pour les allocations dynamiques (les allocation par `malloc`)

- la pile (user stack) pour stocker les variables définies dans les fonctions.

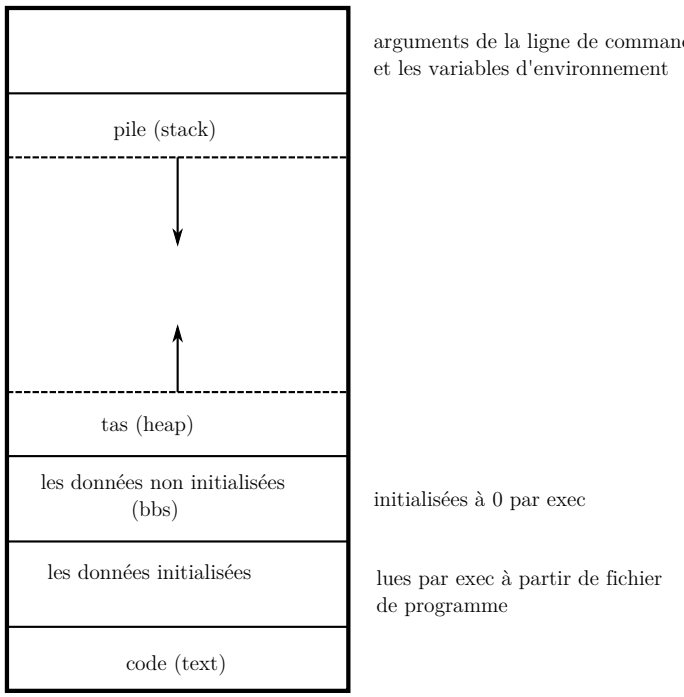


Figure 1: Les segments d'un programme C. Les flèches montrent comment grossissent la pile et le tas.

La commande `size` permet d'afficher la taille de différents segments. Par exemple

```
size copier
  text    data     bss     dec     hex filename
  1172     272       8    1452     5ac copier
```

permet de voir que pour l'exécutable `copier` la taille de segment de code est 1172, segment de données 272, segment bss 8, total 1452 (tout en decimal).

4 Création de processus

Pour voir tous les processus la commande `ps axl`

Création d'un nouveau processus:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void)
```

Tout processus (sauf 0) est créé avec `fork()`

Le processus fils: exécute le même programme que le processus père, avec les mêmes données (copie de données), la même pile d'appel (une copie de la pile du père), le même tas et il possède une copie de la table de descripteurs de fichiers hérité de son père.

A noter que les descripteurs de fichiers ouverts avant fork pointent toujours vers les mêmes entrées dans la table de fichiers ouverts, donc le changement de la position courante dans le fichier par un de deux processus change la position courante dans l'autre.

Les différences entre le processus fils et le processus père:

1. chez le fils `fork()` renvoie 0, chez le père il renvoie le *pid* du fils,
2. le processus enfant a son propre *pid* différents de tous les *pid* et *gid*,
3. le *ppid* de l'enfant initialisé avec le *pid* de père,
4. les mesures de temps consommé sont initialisées à 0 pour l'enfant,
5. les verrous posés par le père ne sont pas hérités,
6. si une minuterie est activé `unsigned int alarm(unsigned int secondes)` elle est désactivée chez le fils,
7. chez le fils l'ensemble de signaux pendants est initialisé à l'ensemble vide.

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void)
```

La fonction `vfork()` fait la même chose que `fork()` mais ne fait pas de copie de données, les données sont partagées entre le processus père et le processus fils. L'utilisation de `vfork()` appropriée uniquement pour faire immédiatement `exec()`

5 Primitives de recouvrement

Il y a six primitives de recouvrement. Différences entre ces primitives:

1. manière de faire passer les arguments de `main()`:
   - soit une liste `l` (`execl, execle, execlp`
   - soit un vecteur `v` (`execv, execve, execvp`),
2. la manière utilisée pour trouver le fichier à charger:

- soit en utilisant la variable `PATH` de l'environnement, ce que indique la lettre `p` dans le nom de la fonction, c'est le cas de primitives `exelp()` et `execvp()`,
- soit relativement au répertoire de travail (ou une référence absolue),

3. l'environnement du processus après le recouvrement, dans le cas de `execve()` et `execle()` un nouvel environnement se substitue à l'ancien.

Les primitives de la famille exec ne retournent jamais en cas de succès. En cas d'échec la valeur de retour est -1.

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ... /*, (char *)NULL*/);
int execv(const char *path, char *const argv[]);
int execle(const char *path,
    const char *arg0, ... /*, (char *)NULL, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)NULL */);
int execvp(const char *file, char *const argv[]);
```

Exemples: Pour remplacer le code actuellement exécuté par le code de `ls`:

```
execl(''/bin/ls'',''ls'',''-l'',''/'',(char *)NULL);
```

`execlp()` se comporte de la même manière sauf que pour résoudre les les références relatives on utilise la variable `PATH`. Par exemple si `PATH` contient `.:/bin:/usr/bin` on pourra faire

```
execlp(''ls'',''ls'',''-l'',''/'',(char *)NULL);
```

La même chose avec le vecteur d'arguments:

```
char *tab[]={"ls", "-l", "/", NULL};
execvp("ls",tab);
```

ou avec un vecteur alloué dynamiquement:

```
char **tab=malloc( 4 * sizeof(char *));
tab[0]="ls";
tab[1]="-l";
tab[2]="/";
tab[3]=NULL;
execvp("ls",tab);
```

5.1 Attributs d'un processus après le recouvrement

(1) **propriétaire effectif** si le bit *set-uid* est positionné sur le fichier chargé le propriétaire de ce fichier devient propriétaire effectif du processus.

(2) **groupe effectif** si *set-uid* est positionné alors le groupe propriétaire de fichier chargé devient groupe propriétaire effectifs du processus,

(3) le *handler* par défaut est installé pour les signaux captés,

(4) **descripteurs** si le bit `DF_CLOEXEC` de fermeture automatique en cas d'exec est positionné par `fcntl()` alors ce descripteur sera fermé. Les autres descripteurs restent ouverts.

6 Terminaison de processus

Le processus se termine soit à sa demande soit à la réception d'un signal provoquant sa terminaison. Le processus qui se termine devient *zombie*, l'information sur la terminaison est enregistrée dans le bloc de contrôle est reste disponible jusqu'à la consultation par le processus père avec les primitives `wait` et `waitpid`

```
#include <unistd.h>
void exit(int valeur)
vois _exit(int valeur)
```

provoque

1. `fflush()` sur tous les fichiers ouverts pour vider les tampons,
2. suppression de fichiers créés avec `tmpfile()`
3. la fonction appelle toutes les fonctions enregistrées avec `atexit()` dans l'ordre inverse d'enregistrement,
4. fait appel à `_exit(valeur)`

Seulement le bit de poids faible de `valeur` peut être récupéré. Deux constantes symboliques `EXIT_SUCCESS` et `EXIT_FAILURE` définies dans `stdlib.h` peuvent être utilisées.

L'appel à `_exit()` provoque

1. fermeture de fichiers et répertoires (sans `fflush()`)
2. envoie de signal `SIGHUP` à tous les processus de groupe si le processus qui se termine est le leader,
3. rattachement de fils au processus 1,
4. si le père est en attente sur `wait()` ou `waitpid()` alors il est réveillé sinon le fils devient zombie. Le signal `SIGCHLD` est envoyé au père.

L'instruction `return` exécutée dans `main()` provoque l'appel à `exit`

L'utilisation de `_exit()` est recommandée si on a fait `exec` qui n'a pas réussi après `vfork()`. Si on fait `exit` cela videra les tampons de fichiers utilisés aussi par le père.

7 Processus zombie et synchronisation père fils

Le processus qui termine devient *zombie*. Il est maintenu dans la table de processus mais ne consomme plus de ressources système. Le processus peut terminer pour deux raisons : (1) terminaison normale par `exit()`, `_exit()` ou `return` de la fonction `main`, dans ce cas le processus retourne le code de retour, (2) une terminaison à la suite de réception d'un signal.

La raison de garder le processus terminé dans la table de processus c'est pour pouvoir récupérer le code de retour ou l'information sur le signal qui a provoqué la terminaison.

Si on exécute

```
ps lx
```

pour voir tous les processus les processus zombies sont marqués par `defunct`.

Les seules informations maintenues dans le bloc de contrôle après la terminaison de processus:

- son code de retour,
- le temps d'exécution dans les modes utilisateur et noyau,
- son identité et l'identité de son père.

Si le père se termine le fils zombie sera attaché au processus 1 qui le supprimera.

Les fonctions `wait()` et `waitpid()` permettent d'obtenir les informations sur le statut d'un processus fils qui est soit terminé soit stoppé.

7.1 wait

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *p_status)
```

(1) Si le processus qui fait appel à `wait()` n'a pas de fils la fonction retourne -1 et `errno=ECHILD`.

(2) si le processus possède un fils zombie alors la fonction retourne l'identité d'un fils zombie, si `p_status` n'est pas NULL alors `p_status` reçoit les informations sur la terminaison. Le fils zombie est supprimé de la table de processus.

(3) Si le processus possède de fils mais aucun fils zombie alors le processus est bloqué

- en attendant qu'un fils devient zombie,
- ou qu'il reçoit un signal. Dans ce deuxième cas la valeur de retour est -1 et `errno` est `EINTR`.

7.2 waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *p_status, int options)
```

permet de tester la terminaison d'un processus appartenant au même groupe de processus que le processus donné. Le paramètre `options` indique si l'attente est bloquante ou non.

Le paramètre `pid` permet de spécifier le processus attendu:

valeur `pid`	attente de la terminaison
< -1	d'un processus quelconque dans le groupe de processus `|pid|`
`(pid_t)`-1	d'un processus fils quelconque (équivalent à `wait()`)
0	d'un processus fils quelconque du même groupe de processus que l'appelant
> 0	du processus fils d'identité `pid`

Le paramètre `options` de `waitpid` est une combinaison bit à bit de valeurs suivantes :

`WNOHANG`	appel non bloquant
`WUNTRACED`	processus concerné est stoppé et cette information n'est pas encore transmise depuis que le processus est stoppé

Si `wait()` ou `waitpid()` retournent parce que l'état de processus fils est disponible ces deux fonctions retournent le pid du processus fils.

Dans le cas contraire les valeurs de retour de `wait()` et `waitpid()` :

- -1 en cas d'erreur,
- 0 en cas d'échec, l'échec signifie que, en mode non bloquant, le processus fils concerné existe mais il n'est ni terminé ni stoppé.

Si l'appel à `wait()` ou `waitpid()` est réussi alors l'entier `*p_status` permet de connaître la raison de terminaison de processus fils : terminaison par `exit()` ou terminaison à la réception d'un signal.

Rappel: Dans le code de retour donné à l'appel de `exit()` juste l'octet de poids faible compte, donc les codes de retour valables sont entre 0 et 255.

Si le processus fils se termine à cause d'un signal alors `*p_status` contient le numéro de signal augmenté de 128 si le fichier `core` a été créé.

Les macro-fonctions suivantes permettent interroger la valeur de `p_status` :

`WIFEXITED`	non nulle si le fils s'est terminé normalement
`WIFSIGNALED`	non nulle si le fils terminé à cause d'un signal
`WIFSTOPPED`	non nulle si le fils stoppé (`waitpid()` avec option `WUNTRACED`

Pour plus de précision on utilise les macro-fonctions suivantes :

`WEXITSTATUS`	donne le code de retour pour le processus terminé normalement
`WTERMSIG`	si le retour de la macro-fonction `WIFSIGNALED` est non nul alors cette macro-fonctions donne le numéro du signal
`WSTOPSIG`	si la valeur de retour de la macro `WIFSTOPPED()` est non nulle `XSTOPSIG` donne le numéro du signal qui a stoppé le processus.

Exemple On attend la terminaison d'un fils et on récupère la valeur exit de ce fils (s'il a terminé par `exit`):

```
pid_t fils;
int status, code_ret;
.....
fils=wait(&status);
if( fils < 0 ){
  /* erreur ou signal */
}
else{
  /* fils zombie supprime*/
  if( WIFEXITED(status) ){ /*le fils a termine par exit*/
     code_ret=WEXITSTATUS(status); /*recuperer le code
                                    * d'exit de fils*/
  ....
  }
}
```

8 Job control

Notion de session et de groupe de processus.

`ctr-z` stop a job (in terminal)
`ctr-q` start
Envoyer le signal stop

```
kill -SIGTSTP processus
```

Envoyer le signal “continuer exécution”

```
kill -SIGCONT processus
```

9 Duplication de descripteur avec dup() et dup2()

Le mécanisme de redirections des entrées et sorties standard est possible grâce à l'utilisation homogène de descripteurs de fichiers et s'applique non seulement aux fichiers réguliers mais aussi aux tubes permettant une communication entre les processus.

Pour cela nous avons besoin une opération de duplication de descripteur. Cette opération crée un nouveau descripteur dans la table de descripteurs d'un processus, le nouveau descripteur pointe vers le même entré dans la table de fichiers ouverts.

```
#include <unistd.h>
int dup(int descripteur)
```

L'argument de `dup()` un descripteur de fichier valable. Cette primitive cherche dans la table descripteurs le plus petit descripteurs qui n'est pas utilisé (qui reste disponible) et et copie le contenu de descripteur passé en arguments vers le descripteur trouvé. La fonction retourne le nouveau descripteur ou -1 en cas d'échec.

Le compteurs de descripteurs ouverts dans la table de fichiers ouverts est incrémenté de 1.

Le compteurs de descripteurs ouverts dans la table de fichiers ouverts est incrémenté de 1. Un exemple est presenté sur la figure 9.

```
#include <unistd.h>
int dup2(int d1, int d2)
```

La primitive `dup2()` ferme le descripteurs `d2` s'il est ouvert et en fait le synonyme de `d1`. La primitive retourne -1 en cas de problèmes et `d2` sinon.

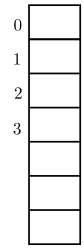
Après l'appel réussi les descripteurs `d2` et `d1` pointent vers la même entrée de la table de fichiers ouverts et le compteur de descripteurs pour ce fichier est augmenté de 1.

- Si `d2<0` ou `d2>=MAX_OPEN` `dup2()` retourne -1 et `errno=EBADF`.
- Si `d1` est un descripteur valide et égale à `d2` alors la fonction `dup2()` retourne `d2` sans le fermer (donc `dup2()` n'a pas d'effet.)
- Si `d1` n'est pas un descripteur valide alors `dup2()` retourne -1 mais ne ferme pas `d2`.

Le nouveau descripteur alloué par `dup()` et `dup2()` possède les mêmes verrous que l'ancien.

```
#include <fcntl.h>
int fcntl(int filedesc1, int cmd, ...)
```

tables de descripteurs



10 Communication par les tubes

Il existe deux types de tubes : anonymes et nommés. Les deux types de tubes ont plusieurs caractéristiques communes.

(1) Chaque tube est représenté par un i-node, donc l'utilisation de tubes ressemble à l'utilisation de fichiers, en particulier dans les programmes les tubes sont accessibles au moyen de descripteurs.

(2) Un tube possède deux extrémités, représentées par deux descripteurs différents : un descripteurs utilisé pour la lecture et l'autre pour écriture. Donc les tubes réalisent une communication unidirectionnelle.

(3) On peut utiliser le mécanisme de duplication de descripteurs sur les tubes. Donc il est possible d'avoir plusieurs descripteurs à l'entrée et à la sortie d'un tube.

(4) Les fils hérite les descripteurs de tube de son père, c'est le mecanisme utilisé pour instaurer une communication entre père et fils.

(5) Les tubes utilisent le mode fifo, l'ordre de lecture est le même que l'ordre d'écriture. L'information est toujours écrite à une extrémité et lue à l'autre. L'information lue est supprimée du tube. Il n'est pas possible de changer la position dans le tube avec `lseek()`

(6) Le tube a une capacité limitée, donc si le tube est plein on ne peut plus écrire.

(7) Si le nombre de lecteurs est nul (aucun descripteur ouvert pour la lecture) l'écrivain essayant écrire dans le tube reçoit le signal `SIGPIPE`.

(8) Si le nombre d'écrivains est nul (le nombre de descripteurs ouverts pour l'écriture dans le tube est nul) alors le lecteur détecte la fin de fichier pendant la tentative de lecture.

Par défaut on utlise les tubes en mode bloquant:

- l'écrivain essayant écrire dans le tube plein est bloqué en attendant qu'un lecteur lit dans le tube en libérant la place.
- le lecteur essayant lire dans un tube vide attend qu'un écrivain y écrit.

Cependant on peut changer le mode de bloquant vers non bloquant.

10.1 Tubes anonymes

Le tube a le compteur de références nul (ces tubes n'apparaissent pas dans un répertoire). Par conséquent un tube anonyme existent (le i-node correspondant existe) s'il y a des processus avec les descripteurs ouverts sur ce tube.

Si le compteur de descripteurs sur le tube anonyme est nul alors le tube sera automatiquement supprimé.

Les premiers descripteurs d'un tube sont créés à la création du tube avec `pipe()`, ensuite d'autres descripteurs peuvent être créés avec le mécanisme de duplication de descripteurs.

Les descripteurs de tubes sont hérités par les processus fils comme d'autres descripteurs de fichiers.

Si le processus a fermé tous les descripteurs d'un tube anonyme soit pour la lecture soit pour l'écriture il lui est impossible d'en acquérir des nouveaux descripteurs pour le même mode (lecture ou écriture).

10.2 Création d'un tube anonyme

```
#include <unistd.h>
int pipe(int desc[2])
```

`desc` est un vecteur de deux entiers. Après l'appel `desc[0]` donne le descripteur ouvert en lecture tandis que `desc[1]` donne le descripteur ouvert en écriture.

Si l'appel réussi alors `pipe()` renvoie 0. En cas d'échec `pipe()` renvoie -1 est

- `errno=EMFILE` – si la table de descripteurs du processus est pleine,
- `errno==ENFILE` – si la table des fichiers ouverts du système est pleine.

Exemple de création d'un tube :

```
int tube[2];

if( pipe(tube) == -1 ){
    perror("creation de tube");
    exit(EXIT_FAILURE);
}
/* maintenant tube[0] - le descripteur d'entree
              tube[1] - le descripteur de sortie
*/
```

10.3 Lecture d'un tube

La lecture s'effectue en utilisant le descripteur avec la primitive `read()` (la même qui est utilisée pour la lecture de fichiers ordinaires) :

```
char buf[TAILLE_BUFF];

int nb_lu = read(tube[0], buf, TAILLE_BUF);
```

Si le tube n'est pas vide et contient `n>0` caractères alors le primitive extrait min(`n,TAILLE_BUF`) caractères et les mets dans `buf`. `read` renvoie le nombre de caractères lus.

Si le tube est vide

- et le nombre d'écrivains est nul, on détecte la fin du fichier, c'est-à-dire `read()` retourne 0.
- si le nombre d'écrivains n'est pas nul et la lecture est bloquante (ce qui est le cas par défaut) alors le processus est mis en attente jusqu'à ce que le tube ne soit pas vide ou qu'il n'y ait plus d'écrivains,
- si le nombre d'écrivains est non nul et la lecture est non bloquante alors le retour immédiat avec la valeur -1 et `errno==EAGAIN`.

Le problème d'inter-blocage:

```
#include <stdlib.h>
#include <unistd.h>
...


int tube[2];
#define BSIZE = 100
char buf[BSIZE];
ssize_t nbytes;
int status;


status = pipe(tube);
if (status == -1 ) {
    /* traiter erreur de pipe() */
    ...
}


switch (fork()) {
case -1: /* traiter erreur de fork() */
    break;

case 0:  /* fils - lit du tube */
    close(tube[1]);                      /* fils n'ecrit pas,
                                          * fermer le descripteur
                                          * d'ecriture */

    while( (nbytes = read(tube[0], buf, BSIZE) ) > 0){   /* lire dans le tube*/
                /* traitement */
    }

    close(tube[0]);                      /* fils termine */
    exit(EXIT_SUCCESS);


default:  /* processus pere - ecrit dans le tube */
    close(tube[0]);                      /* le pere ne lit pas
                                          * fermer le descripteur de lecture */
    write(tube[1], "Hello world\n", 12);  /* pere ecrit */
    close(tube[1]);                      /* fermer le tube pour que
                                            le fils voie la fin de fichier */
    exit(EXIT_SUCCESS);
}
```

10.4 Écriture dans un tube

L'écriture est réalisée avec la primitive `write()`

```
nb_ecrit=write(tube[1], buf, n);
```

Si le nombre d'octets à écrire est `<PIPE_BUF` l'écriture est garantie atomique (les caractères écrits dans le tube d'un seul coup). Si le nombre de caractères à écrire est supérieur à `PIPE_BUF` l'opération peut-être décomposé par le systèmes.

Si le nombre de lecteur dans le tube est nul alors la tentative d'écriture provoque le signal `SIGPIPE` qui, s'il n'est capté, termine le processus écrivain.

Si le nombre de lecteurs est non nul alors

- si l'écriture est bloquante le retour de `write()` n'aura lieu que quand tous les octets sont écrits. Si `n<=PIPE_BUF` alors l'écriture sera atomique. Le processus peut être endormi s'il n'y a pas assez place libre dans le tube.
- si l'écriture est non bloquante
  - si `n>PIPE_BUF` le retour est un nombre de caractères écrits (inférieur à `n`) ou -1 et `errno==EAGAIN`
  - si `n<=PIPE_BUF` et il y a au moins `n` places libres dans e tube alors une écriture atomique de `n` octets est réalisée,
  - si `n<=PIPE_BUF` et il n'y pas `n` places libres dans le tube alors rien n'est écrit, `write()` retourne -1 et `errno==EAGAIN`.

11 Tubes nommés

Un tube nommé peut être créé depuis un interpréteur shell avec

```
mkfifo reference
```

Par exemple

```
mkfifo toto
ls -l|grep toto
prw-r--r-- 1 zielonka zielonka      0 2010-12-06 18:16 toto|
```

Nous pouvons voir que `mkfifo` a créé un fichier spécial `toto` et type de fichier est `p` (*pipe*). Faire `man mkfifo` pour voir les options de `mkfifo`.

Depuis un programme C la création d'un tube nommé s'effectue avec

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *chemin, mode_t droits);
```

où `chemin` donne le chemin d'accès au tube et `droits` les droits d'accès (comme pour `open()`).

Une fois le tube créé on peut y accéder en l'ouvrant avec `open()`, de la même façon que pour les fichiers ordinaires. Les seuls modes possibles sont `O_RDONLY` et `O_WRONLY` Par défaut `open()` est bloquant :

- une demande d'ouverture en lecture est bloquente en absence d'écrivain,
- une demande d'ouverture en écriture est bloquante s'il n'y pas de lecteurs.

Donc le processus peuvent se synchroniser en utilisant le tube, un en lecture et l'autre en écriture.

L'ouverture en mode non bloquant est réalisée avec le drapeau `O_NONBLOCK`

Ouverture non bloquante en lecture réussit immédiatement même ebsence d'écrivains. Les opérations de lecture qui suivent restent non bloquantes (jusqu'à le changement de mode avec `fcntl()`).

Ouverture non bloquante en écriture sans lecteur échoue. La valeur de retour de `open()` est -1 avec `errno==ENXIO`

Rappel : changement de mode : bloquant → non bloquant.

```
#include <unistd.h>
#include <fcntl.h>
fcntl(descripteur, F_SETFL, O_NONBLOCK);
```

retourne une valeur `int` > 0 si l'appel réussi.

12 setuid et setgid

Propriétaire réel du processus – celui qui a lancé le processus, ou plus exactement c'est le propriétaire du processus qui a lancé notre processus avec `exec`.

Le propriétaire effectif – c'est

- soit le propiriétaire réel (dans la plupart de cas),
- soit le propriétaire de fichier exécutable si `setuid` bit est positionné sur le fichier exécutable.

```
> ls -l |grep writer*

-rwxr-xr-x 1 zielonka zielonka 10161 2010-12-08 17:19 writer
-rw-r--r-- 1 zielonka zielonka   9▰9 2010-12-07 01:1▰ writer.c

> chmod u+s writer*
> ls -l |grep writer*

-rwsr-xr-x 1 zielonka zielonka 10161 2010-12-08 17:19 writer
-rwSr--r-- 1 zielonka zielonka   9▰9 2010-12-07 01:1▰ writer.c
```

La lettre `s` ou `S` indique que `setuid` bit est positionné, `s` indique que setuid est positinné sur un fichier exécutable, `S` indique que le fichier n'est pas exécutable mais setuid est positionné quand même.

Les bits setuid et setgid sont utlisé quand on charge un exécutable avec `exec` pour déterminer le propriétaire effectif de processus.