LES PROCESSUS

Licence de ce document

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	20 février 2005	Mise en ligne du document	© Frédéric Lang (frederic-lang@fr.fm)

SOMMAIRE

I) INTRODUCTION

Le système ***Unix*** étant ***multi-tâches***, il implémente des primitives de traitement des processus.
Ces primitives permettent de :

- ✓ obtenir certaines informations sur les processus
- ✓ créer et gérer le déroulement des processus
- ✓ synchroniser des processus
- ✓ faire communiquer ensemble plusieurs processus

II) PRIMITIVES GENERALES

Les deux primitives ***getpid()*** et ***getppid()*** permettent d'obtenir respectivement le numéro courant ***pid*** et celui du père ***ppid*** du processus qui appelle cette primitive.
Les primitives ***getuid()***, ***getgid()***, ***geteuid()*** et ***getegid()*** permettent de récupérer respectivement le numéro :

- ✓ de l'utilisateur qui a lancé le programme (***uid***)
- ✓ du groupe qui a lancé le programme (***gid***)
- ✓ des droits d'utilisateur du processus (***euid***) qui correspondent par défaut à l'***uid***, sauf si le ***setuid*** est positionné
- ✓ des droits d'utilisateur du processus (***egid***) qui correspondent par défaut au ***gid***, sauf si le ***setgid*** est positionné

```
#include <unistd.h>
int getpid ();
int getppid ();
int getuid ();
int getgid ();
int geteuid ();
int getegid ();
```

Valeur renvoyée (int) :

- ✓ un entier représentant le numéro demandé

Les primitives ***setuid()***, ***setgid()***, ***seteuid()*** et ***setegid()*** permettent de modifier respectivement le numéro :

- ✓ de l'utilisateur qui a lancé le programme (***uid***)
- ✓ du groupe de l'utilisateur qui a lancé le programme (***gid***)
- ✓ des droits d'utilisateur du processus (***euid***)
- ✓ des droits d'utilisateur du processus (***egid***)

```
#include <unistd.h>
int setuid (int id);
int setgid (int id);
int seteuid (int id);
int setegid (int id);
```

Explication des paramètres :

- ✓ int id : valeur du nouveau numéro

Valeur renvoyée (int) : un entier représentant le numéro demandé

Cependant l'emploi de ces fonctions est en fait limité aux processus dont le propriétaire est le super utilisateur.

La primitive ***nice()*** permet de modifier la priorité du processus appelant.

```
#include <unistd.h>
int nice (int n);
```

Explication des paramètres :

✓ int n : valeur d'incrément de la priorité. La priorité d'un processus peut varier de 1 (ultra-prioritaire) à 100 (ralenti maximum). Lors du lancement d'un processus, sa priorité se trouve généralement autour de 80. Seul, le super utilisateur ***root*** peut utiliser une valeur *n* négative entraînant un accroissement de la priorité.

Valeur renvoyée (int) : Valeur de la nouvelle priorité du processus.

La primitive ***sleep()*** suspend l'exécution du processus appelant pendant un temps donné.

```
#include <unistd.h>
int sleep (int sec);
```

Explication des paramètres :

- ✓ int sec : temps en secondes durant lequel le processus doit "s'endormir".

III) GESTION DES PROCESSUS

1) Programmation parallèle

La primitive ***fork()*** permet la création dynamique d'un nouveau processus s'exécutant de manière concurrente avec le processus qui l'a créé. Le processus ***fils*** est la copie exacte du processus ***père*** et possède la plupart de ses attributs

- ✓ même propriétaire et groupe
- ✓ mêmes noms de variables avec leurs valeurs telles qu'elles étaient avant le ***fork()***
- ✓ même descripteurs de fichiers ouverts (avec mêmes positions des pointeurs de fichier)

```
#include <unistd.h>
int fork (void);
```

Valeur renvoyée (int) : Valeur permettant de différencier le processus fils du processus père.

- ✓ Elle est égale à 0 dans le processus fils
- ✓ Elle est égale au PID du processus fils dans le processus père

Exemple :

```
#include <unistd.h>
#include <errno.h>
main()
{
        int pid;

        pid=fork();
        switch (pid)
        {
                case (-1) : /* Erreur de fork */
                        perrror("Erreur de fork()");
                        break;
                case 0 : /* Fils */
                        printf("\tJe suis le fils %d\n", getpid());
                        break;
                default : /* Père */
                        printf("Je suis le père %d et du processus %d\n", getpid(), pid);
        }
        /* A partir de ce point coexistent les deux processus */
        printf("Je suis le processus %d\n", getpid());  /* Exécuté par chacun */
}
```

La primitive ***exit()*** met fin à l'exécution d'un processus. Elle renvoie au processus père un code de retour utilisable par ce dernier.

```
#include <unistd.h>
int exit (int n);
```

Explication des paramètres :

✓ int n : code de retour à renvoyer au processus père (la valeur de ce nombre ne doit pas être codé sur plus de 8 bits)

Les primitives ***wait()*** et ***waitpid()*** provoquent la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine. Il est possible cibler le processus fils avec ***waitpid()***.

✓ Si l'un des fils s'est terminé avant l'appel de ***wait()***, le retour de la fonction est immédiat.

✓ S'il n'y a pas de fils, le retour est aussi immédiat et la fonction renvoie ***–1***.

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

Explication des paramètres :

✓ pid_t pid : permet de cibler le processus attendu de la façon suivante :
- pid > 0 : le processus dont le ***pid*** correspond au nombre demandé
- pid = 0 : tout processus fils situé dans le groupe du père
- pid = (-1) : tous les processus fils
- pid < -1 : tous les processus fils dont le groupe correspond à la valeur absolue du nombre demandé

✓ int *retour : pointeur vers une variable ***int*** (2 octets) qui récupèrera la cause de la fin du processus fils :
- processus fils terminé par ***exit(n)*** : la valeur ***n*** se trouve dans l'octet de poids fort de la variable
- processus fils terminé par ***kill(n)*** : la valeur ***n*** se trouve dans les **septs derniers bits** de l'octet de poids faible de la variable (le premier bit de cet octet étant à "1" s'il y a création d'un fichier core).

✓ int options : combinaison de bits modifiant l'action de la fonction. Les valeurs possibles sont :
- WUNTRACED : si le processus est arrêté par un signal, la fonction s'en apercevra
- WCONTINUED : si le processus fils est relancé par un signal, la fonction s'en apercevra
- WNOHANG : mode non-bloquant (utilisé pour palper un processus).
- WNOWAIT : l'information récupérée par la fonction n'est pas effacée dans le processus fils. La fonction peut donc être appelée de nouveau et traitera à nouveau le processus fils précité.

Valeur renvoyée (int) :

✓ Numéro du processus fils qui vient de se terminer

✓ 0 en mode WNOHANG si le processus demandé n'est pas terminé.

Pour interpréter la valeur de retour, il est nécessaire de savoir manipuler une variable octet par octet. Les opérateurs "bit à bit" sont ici nécessaires.
Exemple :

```
if ((retour & 0xFF00) != 0)      /* Teste valeur premier octet différente de 0 */
if ((retour & 0x00FF) != 0)      /* Teste valeur second octet différente de 0 */
(retour >> 8)                    /* Valeur n correspondant au "exit(n)" du fils */
(retour & 0x007F)                /* Valeur n correspondant au "kill(n)" */
```

Des macro définitions ont été implémentées. Pour des raisons de portabilité et d'évolution future, l'utilisation de ces macros est recommandée en lieu et place de l'exemple précédent

Macro Définition	***Interprétation***
WIFEXITED(n)	valeur non nulle si le processus fils s'est terminé par "exit()"
WEXITSTATUS(n)	fournit le code de retour du processus si le processus s'est terminé normalement
WIFSIGNALED(n)	valeur non nulle si le processus fils s'est terminé par "kill()"
WTERMSIG(n)	fournit le numéro du signal ayant provoqué la terminaison du processus
WIFSTOPPED(n)	valeur non nulle si le processus est stoppé (primitive waitpid avec option WUNTRACED)
WSTOPSIG(n)	fournit le numéro du signal ayant stoppé le processus

2) Primitives de la famille *exec()*

Les primitives de la famille ***exec()*** permettent de remplacer un processus par un autre. Toutes les données, variables environnement du premier processus sont perdus. On parle alors de **recouvrement**. Cependant, les fichiers ouverts avant l'appel restent ouverts.
Il n'y a pas qu'une primitive ***exec()*** mais plusieurs qui diffèrent en fonction de la syntaxe utilisée et du résultat souhaité. Celles-ci sont regroupées en trois familles :

✓ fonction de base : elle reçoit le nom complet du programme à lancer avec ou sans argument

✓ fonction avec chemin d'accès : le nom du programme sera cherché dans la variable ***PATH*** de l'environnement du programme appelant

✓ fonction avec contexte de travail : le processus lancé recevra en plus un environnement de travail qui pourra avoir été construit au préalable

De plus, chaque famille est subdivisée en deux groupes :

✓ les paramètres de travail sont listés un à un dans l'appel

✓ les paramètres de travail sont envoyés sous forme de tableau de paramètres

```
#include <unistd.h>
int execl (char *prog, char *arg0, char *arg1, …, char *argn);
int execv (char *prog, char *argv[]);
int execlp (char *prog, char *arg0, char *arg1, …, char *argn);
int execvp (char *prog, char *argv[]);
int execle (char *prog, char *arg0, char *arg1, …, char *argn, char
*envp[]);
int execve (char *prog, char *argv[], char *envp[]);
```

Explication des paramètres :

✓ char *prog : nom du programme à lancer

✓ char *arg0, arg1, …, argn : arguments à passer au programme sous forme d'éléments séparés. Ceux-ci seront reçus dans le paramètre ***argv***. de la fonction ***main()*** du programme. Ne pas oublier le zéro final

✓ char *argv[] : arguments à passer au programme sous forme de tableau statique ou dynamique. Ceux-ci seront reçus dans le paramètre ***argv***. de la fonction ***main()*** du programme. Ne pas oublier d'avoir inclus le zéro final dans le tableau.

✓ char *envp[] : environnement que doit recevoir le programme. Celui-ci sera reçu dans le paramètre ***envp*** de la fonction ***main()*** du programme.

Exemple :

```
execl("/bin/ls", "ls", "-l", "/etc/passwd", "/etc/group", NULL);
```

IV) SYNCHRONISATION DES PROCESSUS

Sous UNIX, les processus disposent d'espace mémoire indépendants assurant la sécurité mutuelle des codes et des données. Néanmoins ce système interdit toute possibilité de synchronisation entre processus et, bien entendu, de communications entre eux.
Néanmoins, UNIX implémente certains dispositifs qui permettent de réaliser la coordination entre les processus au cours de leurs exécutions.

1) Les signaux

a) Généralités

Des événements extérieurs ou intérieurs au processus peuvent survenir à tout moment et provoquer une interruption de l'exécution du processus en cours. Ces événements peuvent être d'origine "physique" (coupure de courant, lecteur non prêt, etc.) ou logique.
Lors de la réception d'une interruption, le noyau reconnaît l'origine de celle-ci, sauve le contexte du processus actuel et provoque l'exécution d'une routine appropriée ***handler***. Le mécanisme est semblable à celui des interruptions sous DOS.
La technique des signaux est utilisée par le noyau pour prévenir un processus de l'existence d'un événement extérieur le concernant. Elle correspond à l'utilisation, dans d'autres systèmes d'exploitation, du principe des interruptions logicielles. L'origine d'un signal peut être variée, elle peut être d'origine matérielle, système ou logicielle. Ce peut être :

- ✓ la fin d'exécution d'un processus fils
- ✓ l'utilisation d'une instruction illégale dans le programme
- ✓ la réception d'un signal d'alarme
- ✓ une demande d'arrêt de la part du processus
- ✓ etc.

Le noyau prévoit une réaction par défaut à chaque signal (routine ***handler***). Cette réaction est en général d'arrêter le processus recevant ce signal, mais le comportement du processus à la réception d'un signal peut être configurable. Le processus peut ainsi se prémunir contre l'effet de certains signaux.

b) Les signaux

Les différents signaux sont représentés par des numéros auxquels sont associés des noms symboliques utilisés par les compilateurs.

Nom	***N°***	***Rôle***	***Core***	***Ignoré***
SIGHUP	1	émis au processus attaché à un terminal, lorsque celui-ci est déconnecté (HangUP)		
SIGINT	2	émis au processus attaché au terminal sur lequel on frappe <DEL> ou <SUPPR>		
SIGQUIT	3	idem pour la touche <QUIT>	**X**	
SIGILL	4	envoyé au processus qui tente d'exécuter une instruction illégale	**X**	
SIGTRAP	5	envoyé après chaque instruction. Utilisé par les programmes de mise au point	**X**	
SIGIOT	6	émis en cas de problème matériel	**X**	
SIGEMT	7	utilise pour les calculs en virgule flottante		
SIGFPE	8	erreur sur les flottants	**X**	
SIGKILL	9	émis pour tuer le processus		
SIGBUS	10	émis en cas d'erreur sur le bus	**X**	
SIGSEGV	11	émis lors d'une violation de mémoire	**X**	
SIGSYS	12	émis lors d'une erreur dans les paramètres transmis à une primitive	**X**	
SIGPIPE	13	émis lors de l'écriture dans un pipe sans lecteur		
SIGALRM	14	signal associé à l'horloge système		
SIGTERM	15	signal de fin normale d'un processus		
SIGUSR1	16	offert aux utilisateurs		
SIGUSR2	17	idem		
SIGCHLD	18	émis vers le père à la mort du fils		**X**
SIGPWR	19	émis lors d'une coupure de courant		**X**
SIGPOLL	22	événement sélectionné		**X**

core : X, ces signaux génèrent un fichier core
Certains signaux génèrent la production d'une image du contexte du processus lors de son interruption. Cette image est appelée core. Un message "core dumped" est alors envoyé sur l'écran par le système : et un fichier core est créé dans le répertoire de travail. Il peut être lu avec certains outils (débuggeur) pour essayer de comprendre ce qui a pu générer l'interruption.

ignoré : X, ces signaux sont ignorés par défaut
Chaque système UNIX soutient un nombre différent de signaux. De plus les numéros n'ont pas la même signification. Il n'y a que les constantes prédéfinies qui soient portables

c) Fonctionnement d'un signal :

Envoi d'un signal :

Un signal peut être envoyé par le noyau ou un processus. Ce signal est mémorisé dans un champ de la table proc du processus récepteur (on dit que c'est un signal pendant) : chaque signal active un bit particulier de ce champ.

Lorsque le processus récepteur doit être activé ou lorsqu'il réalise un appel système, le scheduler lit ce champ est fait exécuter les routines correspondantes aux différents bits de signaux activés (le signal est dit délivré).

Le moment entre l'émission du signal et sa prise en compte par le processus est donc indéterminé.

Sous System V Release 4 un signal peut être bloqué : sa prise en compte est différée jusqu'à ce que le signal ne soit plus bloqué.

Réception d'un signal :

En règle générale, la réception d'un signal provoque l'arrêt de l'exécution d'un processus avec éventuellement génération d'une image mémoire core et se traduit par l'exécution de la fonction handler associée.

d) Fonctions de gestions des signaux :

La primitive ***kill()*** envoie un signal à un processus . Elle prend en paramètre l'identificateur du processus destinataire et le numéro du signal. Si la valeur PID=0 est utilisée, le signal est envoyé à tous les processus du groupe.

```
#include <signal.h>
int kill (int pid, int sig);
```

Explication des paramètres :

✓ int pid : numéro du processus où sera envoyé le signal. Ce numéro peut avoir quatre groupe de valeurs

- ☞ Une valeur ***n*** positive enverra le signal au processus de pid "n"
- ☞ La valeur ***0*** enverra le signal à tous les processus du même groupe que le processus appelant
- ☞ La valeur ***–1*** enverra le signal à tous les processus tournant sur la machine (sauf le premier "*init*") par ordre de pid décroissant
- ☞ Une valeur ***n*** négative autre que ***–1*** enverra le signal à tous les processus du même groupe que le processus de pid "-n"

✓ int sig : numéro du signal à envoyer. La valeur ***0*** est utilisée pour vérifier que le processus existe.

Valeur renvoyée (int) : 0

La fonction ***signal()*** permet de prémunir un processus contre l'effet principal de la réception d'un signal quelconque qui est d'arrêter son exécution. Cette fonction ira installer dans une zone particulière correspondant au signal attendu une adresse de déroutement (fonction).

```
#include <signal.h>
int signal (int sig, void (*pt_fonc)(int));
```

Explication des paramètres :

- ✓ int sig : numéro du signal devant être reçu
- ✓ void (*pt_fonc)(int) : référence (adresse) de la fonction qui sera exécutée lors de la réception dudit signal. Cette fonction, à charge du programmeur, doit impérativement être prévue pour recevoir un ***int*** (qui sera, dans la fonction, le signal reçu) et ne doit rien renvoyer (***void***). Deux valeurs particulières peuvent êtres utilisées pour ce paramètre :
  - ☞ SIG_IGN : le signal est purement et simplement ignoré (mais il est quand même acquitté). Ce paramètre est utile si la réception d'un signal ne donne pas lieu à l'appel d'une fonction particulière
  - ☞ SIG_DFL : restitue au processus son comportement par défaut (mourir)

Valeur renvoyée (int) : Valeur précédente de déroutement.

Le signal 9 (SIGKILL) ne peut pas être ignoré ni détourné.

Remarque **spécifique** à l'environnement **System V** :
Lors de la réception d'un signal, la fonction préalablement chargée par la primitive ***signal()*** est alors exécutée. Mais dans le même temps, le comportement par défaut du processus est restauré. Ce qui signifie que lors de la réception d'un nouveau signal, le processus aura retrouvé son comportement original et se terminera.
Pour avoir un aspect permanent, une fonction prévue à la réception d'un signal doit procéder à une réinstallation de son propre code au moyen de la même primitive ***signal()***.

Exemple :

```
#include <signal.h>
void trt_sig(int sig)                    /* Traitement du signal reçu */
{
        signal(sig, trt_sig);            /* Réarmement de la fonction */
        printf("J'ai reçu le signal %u\n", sig);
}

main()                                   /* Fonction principale */
{
        signal(15, trt_sig)              /* Détournement du signal 15 */
        …                                        /* Code quelconque */
}
```

Cette remarque ne s'applique pas à l'environnement **BSD** et un signal détourné une fois le reste en permanence ou jusqu'à ce que son comportement par défaut soit restauré. Il est interressant de noter que la norme **POSIX** qui tend à se mettre en place un peu partout correspond, pour la primitive ***signal()***, à cet environnement.

la primitive ***pause()*** suspend l'exécution du processus qui l'invoque jusqu'à l'arrivée d'un signal, quel qu'il soit. Bien entendu, si le processus n'a pas, au préalable, détourné le signal arrivant, ce processus ne se réveillera que pour mourir.

```
#include <unistd.h>
int pause (void);
```

la primitive ***alarm()*** duplique son code lors de l'appel (tout comme la primitive ***fork()***) mais le code dupliqué n'est pas accessible au programmeur

✓ Le code père (qui a lancé la primitive) rend immédiatement la main (et peut donc faire autre-chose

✓ Le code fils (qui vient d'être généré) attendra un certain temps et, lorsque ce temps sera écoulé, enverra au processus père (processus qui a lancé la primitive) un signal SIGALRM (et pas un autre) ; puis se terminera proprement

```
#include <unistd.h>
int alarm (int sec);
```

Explication des paramètres :

✓ int sec : temps d'attente du fils

Cette primitive peut-être utilisée lorsqu'on veut prévenir un processus de manière automatique (temps écoulé, etc.)

Remarque : lancer une seconde fois cette primitive alors que le temps de la première n'est pas encore écoulé annule le premier appel.

e) Gestion du signal SIGCHLD :

Ce signal est envoyé par tout processus fils à son processus père lorsqu'il meurt. Il faut dans certains cas réaliser une bonne gestion de la réception de ce signal.

Rappel : Il est possible que le processus père, après avoir créé un processus fils, ait continué son exécution sans attendre la terminaison de son fils. Un processus fils restera dans l'état de processus "zombie" aussi longtemps que son père n'aura pas consulté son code de retour.

L'indicateur d'arrivée du signal SIGCHLD est positionné dans le bloc de contrôle des signaux du processus père dès qu'un de ses fils s'est terminé et le reste aussi longtemps que le code de retour d'un de ses fils n'a pas été consulté par l'intermédiaire de la primitive ***wait()***. Il est par conséquent essentiel de gérer correctement la terminaison des processus fils afin de ne pas saturer la table des processus avec des processus zombies. C'est un appel à la primitive ***wait()*** qui bascule l'état du bit de contrôle du signal SIGCHLD, à la différence des autres signaux.
Pour assurer une bonne gestion de la mort des processus fils, et donc d'éviter la création de processus zombies, il est préférable d'utiliser la primitive ***waitpid()***, qui est une extension optimisée de la primitive ***wait()***.

2) LES VERROUS

a) Généralités :

Du fait de l'environnement Multi-Utilisateurs, il est possible que plusieurs d'entre eux tentent d'accéder en même temps à une même ressource (fichier ou imprimante...).
L'accès à un fichier est à considérer comme un accès à une ressource critique dans la mesure où des accès non maîtrisés rendent le contenu d'un fichier aléatoire. Il faut donc mettre au point des systèmes permettant de contrôler ces accès : les verrous sont une première réponse mise à la disposition des programmeurs.
Il faut se rappeler que sous Unix tout pouvant être considéré comme un fichier, la technique des verrous est donc générale.

b) Types de verrous :

UNIX propose diverses possibilités de verrous : il y a les verrous externes, devenus obsolètes, et les verrous intégrés à l'objet à verrouiller . D'autre part, un verrou peut agir sur la totalité du fichier ou seulement sur une partie de celui-ci.

Enfin un verrou peut être :

✓ **impératif** : Il interdit alors effectivement la réalisation de certaines opérations d'E/S. C'est le verrou le plus efficace mais il alourdit le travail du noyau.

✓ **consultatif** : Il faut alors que les processus testent l'existence du verrou. S'ils ne le font pas ils peuvent réaliser les E/S souhaitées mais le verrou est alors inutile.

✓ **partagé/exclusif** : Un verrou partagé (en lecture) est compatible avec des verrous du même type : il vise essentiellement, en lecture, à interdire toute modification du fichier et à empêcher la pose d'un verrou exclusif en écriture .

L'utilisation non contrôlée de verrou peut conduire à des situations de blocage lorsque deux processus ont chacun posé un verrou exclusif bloquant l'autre processus.
Un verrou en lecture ne peut être posé que s'il n'existe pas de verrou en écriture sur le fichier. De même un verrou en écriture ne peut être posé que s'il n'existe aucun verrou sur le fichier.

c) Gestions des verrous par le système :

Les manipulations de verrous se font par l'intermédiaire de la primitive ***fcntl()***. Le système gère une table des verrous au sein du noyau.
Les différents verrous posés sur un fichier donné sont chaînés. Une entrée dans la table des verrous contient les informations suivantes :

- ✓ Type de verrou (écriture/lecture)
- ✓ Caractéristiques de la zone verrouillée (début et longueur)
- ✓ Un pointeur sur l'entrée, dans la table des processus, du processus poseur du verrou
- ✓ Les pointeurs de chaînage des verrous.

La structure ***flock***, définie dans ***fcntl.h***, sert à la réalisation par la primitive ***fcntl()*** des diverses opérations sur les verrous :

```
struct flock {
   short l_type;          /* type de verrou */
   short l_whence;        /* position absolue début du verrou */
   long l_start;          /* position relative début du verrou */
   long l_len;            /* longueur de la zone verrouillée */
   short l_pid;           /* processus propriétaire */
}
```

Explication des éléments :

- ✓ short l-type : type de verrou posé
  - ☞ F_RDLCK : verrou partagé (en lecture)
  - ☞ F_WRLCK : verrou exclusif (en écriture)
  - ☞ F_UNLCK : absence de verrou
- ✓ short l_whence : portée du verrou selon trois paramètres :
  - ☞ 0 le début de la zone verrouillée est au début du fichier
  - ☞ 1 la zone verrouillée commence à la position courante
  - ☞ 2 : la zone verrouillée commence à la fin du fichier
- ✓ long l_start : position de départ relative au champ ***l_whence*** où sera posé le verrou
- ✓ long l_len : longueur de la zone verrouillée. Une valeur spéciale ***0*** indique que le verrou sera posé jusqu'à la fin du fichier
- ✓ short l_pid : numéro du propriétaire du verrou
- ✓ e type de verrou posé

d) Manipulation des verrous :

La primitive ***fcntl()*** permet de réaliser une opération de verrouillage sur un fichier particulier.

```
#include <fcntl.h>
int fcntl (int desc, int op, struct flock *pt_verrou);
```

Explication des paramètres :

- ✓ int desc : descripteur du fichier concerné
- ✓ int op : opération à effectuer. Celles-ci sont au nombre de 3 :
  - ☞ F_GETLK : lecture des caractéristiques d'un verrou existant
  - ☞ F_SETLK : pose ou changement de verrou en mode non bloquant
  - ☞ F_SETLKW : pose ou changement de verrou en mode bloquant
- ✓ struct flock *pt_verrou : pointeur vers une variable de type ***struct flock***. Celle-ci devra avoir été préalablement remplie.

La pose d'un verrou en lecture suppose le descripteur ouvert en lecture (idem pour une écriture).
La fermeture d'un descripteur par un processus implique la levée de tous les verrous posés par ce processus sur le fichier correspondant au descripteur.

Exemple :

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>

main (int argc, char *argv[ ])
{
        int  m, d;
        char c;
        stuct flock verrou;

        d=open("fic ", O_RDWR);
        verrou.l_type=F_WRLCK;
        verrou.l_whence=0
        verrou.l_start=0
        verrou.l_len=0;
        while (fcntl (d, F_SETLK, &verrou) == (-1) && (errno == EACCES || errno == EAGAIN))
        {
                perror("Pose de verrou impossible\n");
                sleep (5);
        }
        printf ("Verrou posé\n");
        while ((m=read (d, &c, 1)) != 0)
        {
                if (c == '# ')
                {
                        lseek (d, -1L, 1);
                        sleep (1);
                        write (d, argv[1], 1);
                        return(0);
                }
                if (m== 0)
                        printf (" Pas d'occurrence du caractère '#'\n");
        }
}
```

3) Dispositif de scrutation

On est parfois amené à vouloir qu'un processus récupère des informations sur un ou plusieurs périphériques du système, via des descripteurs de fichier. La plupart du temps les opérations de lecture et écriture sur ces dispositifs étant bloquantes, il est difficile d'assurer la lecture de tous ces descripteurs dans des délais satisfaisant ce qui est pénalisant en terme de performances.

Il est donc souhaitable que l'on puisse utiliser des primitives permettant au processus,

- ✓ soit de lire alternativement en mode non bloquant sur chaque descripteur
- ✓ soit d'être averti par le système lorsqu'une lecture doit être réalisée sur un descripteur.

a) La primitive poll () :

La structure ***pollfd***, définie dans ***sys/poll.h***, permet la réalisation par la primitive ***poll()*** de multiplexage par un processus

```
struct pollfd {
   int fd;                  /* descripteur du fichier à scruter */
   short events;            /* événements attendu sur le fichier */
   short revents;           /* événement survenus sur le fichier */
}
```

Explication des éléments :

- ✓ int fd : descripteur du fichier à interroger
- ✓ short event : champ contenant les évènements à scruter
- ✓ short revents : champ contenant les évènements survenus

La primitive ***poll()*** permet la réalisation de multiplexage par un processus.

```
#include <sys/poll.h>
int poll (struct pollfd *pt_tabpoll, int nbelem, int temps);
```

Explication des paramètres :

✓ struct pollfd *pt_tabpoll : pointeur vers un tableau de variables de type ***struct pollfd***. Chacune des variables du tableau aura été préalablement remplie.

- ✓ int nbelem : Nombre d'éléments contenus dans le tableau
- ✓ int temps : durée de la scrutation en millisecondes

Valeur renvoyée (int) : nombre de fichiers ayant reçu au moins un évènement attendu (champ ***revents*** non nul).

Les événements sont obtenus comme combinaison des événements suivants :

- ✓ POLLIN : lecture possible de données autres que *hautement prioritaire*
- ✓ POLLRDNORM : lecture possible de données *normales*
- ✓ POLLRDBAND : lecture possible de données *prioritaires*
- ✓ POLLPRI : lecture possible de données *hautement prioritaires*
- ✓ POLLOUT : écriture possible de données
- ✓ POLLWRNORM : *idem* POLLOUT
- ✓ POLLWRBAND : écriture possible de données *prioritaires*
- ✓ POLLERR (champ ***revents*** uniquement) : une erreur est intervenue
- ✓ POLLHUP (champ ***revents*** uniquement) : coupure de ligne
- ✓ POLLINVAL (champ ***revents*** uniquement) : le fichier spécifié n'est pas ouvert

b) Entrées-Sorties asynchrones :

Les processus peuvent demander au noyau de les avertir lorsqu'une lecture ou une écriture est réalisable sur un fichier. Ils recevront alors le signal SIGIO.
Pour pouvoir réaliser cela il faut réaliser les opérations suivantes :

- ✓ Créer un handler pour traiter la réception du signal SIGIO.
- ✓ Armer la réception du signal pour le processus ou le groupe de processus, souhaité. Cette action est réalisée par la fonction fcntl ().
- ✓ Positionner l'option asynchrone pour le processus, en utilisant la primitive ***fcntl()***.

Exemple :

```
#include < signal.h>
#include < stdio.h>

void r_sigio ()                                  /* routine à associer au signal SIGIO */
{
        char buf[80];
        int nb;                                  /* nombre d'octets */

        nb=read(0, buf, 80);
        buf[nb]=0;                               /* transformation en chaîne de caractères */
        printf("le message reçu est : '%s'\n", buf);
}

int main ()
{
        signal(SIGIO, r_sigio);                  /* détournement signal */
        fcntl (0, F_SETOWN, getpid ());

        fcntl (0, F_SETFL, FASYNC);              /* mode E/S asynchrones */

        i=0;
        while (1)                                /* boucle interrompue par SIGIO */
                printf (" i=%d\n", i++);
}
```

V) LES TUBES DE COMMUNICATION

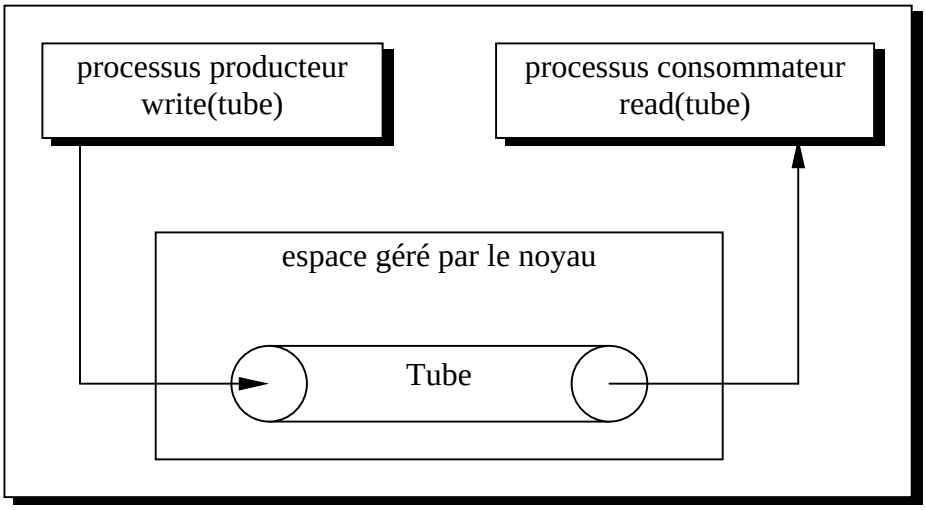
Ce chapitre présente les tubes de communications inter processus. Ce mécanisme est un des différents outils permettant de faire communiquer deux processus entre eux

1) Généralités

Un tube est une zone mémoire limitée (en général de 4Ko) permettant à deux processus de faire transiter des données de l'un vers l'autre. Son mode de fonctionnement est FIFO (First-In, First-Out). C'est à dire que le premier élément à entrer dans le tube sera le premier à en sortir.
Le principe est qu'un processus ira mettre des données dans le tube tandis qu'un second récupèrera les données émises par le premier. La lecture est bloquante tant que le tube est vide, l'écriture est bloquante tant que le tube est plein.
Les tubes sont unidirectionnels. Il faut utiliser deux tubes si, dans l'échange, chaque processus est à la fois producteur et consommateur.
Il y a deux types de tubes :

✓ les tubes "mémoire" : ils sont créés dans la mémoire du processus qui les génèrent. Cela implique que les deux processus qui communiqueront via ce système soient descendants d'un même processus créateur (***fork()***) et que le tube ait été créé avant la duplication pour que les deux process le connaissent.

✓ les tubes "nommés" : ils correspondent à un fichier (inode) dans lequel les processus pourront lire et écrire. Cela implique opération sur périphérique mais deux processus indépendants pourront se transmettre des informations.
Les opérations autorisées sur un tube sont la création, la lecture, l'écriture, et la fermeture.



2) Lecture/Ecriture/Fermeture

La lecture/écriture/fermeture se font dans un tube, quel qu'il soit, par l'intermédiaire des primitives ***read()***, ***write()*** et ***close()*** déjà vues dans la gestion des fichiers.

```
#include <unistd.h>
int read (int desc, char *buf, int nb);
int write (int desc, char *buf, int nb);
int close (int desc);
```

3) Création

a) tubes "mémoire"

La primitive ***pipe()*** a pour fonction de créer le tube en mémoire et ouvrir le tube créé (donc, la primitive ***open()*** est inutile dans un pipe mémoire)..

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

Explication des paramètres :

✓ int p[2] : pointeur vers un tableau de deux entiers qui seront remplis par la fonction. A la fin de l'exécution de celle-ci, on aura :

- p[0] contiendra un descripteur permettant l'accès au tube en lecture
- p[1] contiendra un descripteur permettant l'accès au tube en écriture

Chacun des processus ayant été créé après l'appel à *pipe()*, chacun aura accès aux deux cotés du tube (lecture/écriture). Chacun devra donc commencer par fermer le coté qu'il n'utilise pas ; en particulier le coté d'écriture pour le processus lecteur.
Cela permettra plus tard au noyau, lorsque le processus écrivain aura fini son travail et fermé son coté d'écriture, de détecter que le tube n'a plus aucun processus au coté d'écriture ouvert et d'envoyer alors un caractère "EOF" (-1) à tout processus lecteur éventuel du tube.

b) tubes "nommés"

Ces fichiers peuvent être créés par la primitive ***mknod()***, avec un mode spécial S_IFIFO, ou bien par la primitive ***mkfifo()***. Une fois ces tubes créés, le processus désireux d'écrire ou lire le tube devra utiliser la primitive ***open()***.

```
#include <fcntl.h>
int mknod (char *nom, int droits, int dev);
int mkfifo (char *nom, int droits);
int open (char *nom, int mode[, int droits]);
```

Explication des paramètres :

✓ char *nom : nom du fichier à créer. Tous les répertoires précédant le nom doivent exister et être accessibles (droit "x").

✓ int droits : droits attribués au fichier (utiliser les constantes vues dans la structure ***stat***). Pour la primitive ***mknod()***,il faudra y combiner la constante S_IFIFO.

✓ int dev : variable contenant dans son premier octet le nœud majeur et dans son second octet le nœud mineur du fichier "spécial" que l'on veut créer. Remplir cette variable nécessite de savoir manipuler les bits, ou les ***union*** ; mais cette variable n'a de signification que si la primitive est utilisée pour créer un fichier spécial ("bloc" ou "caractère"), ce qui n'est pas le cas d'un fichier "pipe".

✓ int mode : mode d'ouverture demandé. Celui-ci n'est composé impérativement que d'une des deux constantes suivantes :

- ☞ O_RDONLY : ouverture en lecture seule
- ☞ O_WRONLY : ouverture en écriture seule

et facultativement, de la constante O_NDELAY permettant de demander aux process accédant à un tube vide pour la lecture, ou non encore lu pour une écriture de ne pas être bloqué par ce détail.
Par défaut, pour un tube nommé, l'ouverture est bloquante (mise en sommeil du processus) en lecture s'il n'y a pas de processus écrivain et en écriture s'il n'y a pas de processus lecteur. Le positionnement du bit O_NDELAY supprime ce blocage.

A noter :

✓ La lecture dans un tube vide est bloquante, sauf si O_NDELAY a été positionné. Dans ce cas particulier, le caractère lu est non significatif et la primitive ***read()*** renvoie la valeur 0.

✓ L'écriture dans un tube plein est bloquante, sauf si O_NDELAY a été positionné. Dans ce cas particulier, la primitive ***write()*** renvoie la valeur 0.

Une tentative d'écriture dans un tube nommé dont tous les processus lecteurs ont disparu provoque l'envoi du signal SIGPIPE et donc l'interruption du processus.
Pour un bon fonctionnement des tubes nommés, il faut donc tester la valeur de retour des primitives ***read()*** et ***write()***.

La suppression d'un tube nommé se fait par la primitive habituelle de suppression de fichier ***unlink()***.

```
#include <unistd.h>
int unlink (char *nom);
```

Explication des paramètres :

✓ char *nom : nom du fichier à supprimer

INDEX