Module n°1

PROGRAMMATION SYSTÈME

Auteur : Singodiwirjo Hermantino
Version 1.0 – 13 novembre 2003
Nombre de pages : 23

Paris Academy of Computer Science
SUPINFO Paris
Ecole Supérieure d'Informatique

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com - herman@supinfo.com

Table des matières

1. Introduction

Maintenant que vous avez acquis une bonne connaissance du système GNU/Linux et que vous possédez les bases nécessaires en programmation C, quoi de plus naturel que de doucement passer d'utilisateur/administrateur à développeur ?
La possibilité d'accéder aux sources de la majeure partie des programmes fonctionnant sous Linux vous fourni la plus grande bibliothèque d'exemples possible. Cependant il ne sert à rien de foncer tête baissée dans un amas de lignes de codes sans posséder les clefs nécessaires à leur compréhension.
Nous allons aborder la programmation système en plusieurs modules.
Le premier, celui même que vous lisez en ce moment, vous présente les bases nécessaires à la programmation des processus, à l'accès à l'environnement et à l'exécution de programmes.
Un second module sera consacré à l'ordonnancement des **processus** et à la gestion de **signaux** classiques, Posix.1 et temps réel (Posix.1b) .
Le troisième module décrira les **threads** et les communications inter processus classiques et System V.
Enfin on terminera ce cours avec un module consacré au entrées/sorties avancées (**IO**), et à la programmation réseau.
Il faut bien comprendre que ce cours peut beaucoup apporter, même à un administrateur, car il permet de mieux comprendre le système utilisé.
De plus il s'agit là d'une des forces du systèmes Linux de pouvoir ainsi manipuler des appels systèmes pour certain de très bas niveau (mmap(), fork(), fcntl() ...)

1.1. Qu'est-ce que la programmation système ?

Lorsque vous développez un logiciel plusieurs composants entrent en œuvre. Vous allez tour à tour implémenter les algorithmes nécessaires, l'interface utilisateur et bien d'autres modules. Si votre projet est un peu plus complexe qu'un simple « Hello World » ou même qu'une simple calculatrice basique, vous allez être confronté à de la programmation système.
Par exemple vous voulez que votre programme en exécute un autre, ou bien encore plus, qu'il crée un (ou plusieurs) processus enfant et communique avec eux, il s'agit de programmation système (**IPC** : Inter Process Communication).
Ou bien vous voulez peut-être que votre programme écoute sur plusieurs descripteurs en même temps (clavier, **socket** etc..) et tout cela sans bloquer bien entendu, la encore on touche à la programmation système, et cela sans parler des threads bien sûr.
Nous n'allons pas citer dans cette introduction l'ensemble des sujets englobés dans la programmation système, mais sachez simplement qu'il est essentiel à un développeur de pouvoir comprendre et mettre en œuvre ce type de programmation.

Ceci nous conduira petit à petit, grâce à une compréhension plus profonde du système, à la programmation du noyau Linux lui-même !

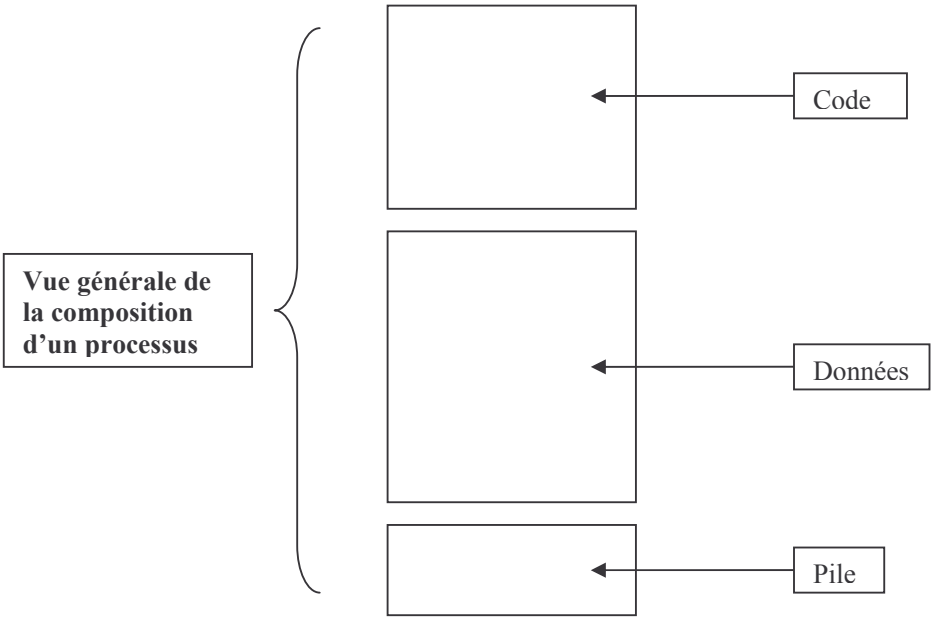
1.2. Pré requis

Pour ne pas se sentir perdu, il est nécessaire d'être relativement alaise avec le langage C, et avec le système GNU/Linux plus généralement.

2. Notion de processus

2.1.Présentation des processus

Du point de vue de l'utilisateur on pourrait définir un processus comme l'instance d'un programme qui s'exécute. Essayons maintenant d'observer et de définir cette entité de plus près.
En fait un processus est composé de **code** (modèle d'octets interprétés par l'unité centrale), de **données** et d'une **pile** :



L'exécution du code est complètement **séquentielle** et il est impossible de faire un saut dans les instructions d'un autre processus.
En fait notre première définition est imparfaite, en effet, non seulement un programme peut utiliser plusieurs processus mais un unique processus peut lancer l'exécution d'un nouveau programme en remplaçant entièrement le code et les données du programme précédent.
Comme nous allons le voire plus en avant, à un instant donné un processus peut être dans différents états. C'est le noyau du système d'exploitation qui est chargé de réguler l'ensemble des processus en leur fournissant équitablement l'accès au microprocesseur. Ce mécanisme de régulation est appelé **ordonnanceur**.

2.2.Identification d'un processus par son PID

Le premier processus existant sur un système Linux est créé au démarrage par le noyau, il s'agit d'*init*.
A partir de ce moment, la seule manière de créer un processus est d'utiliser l'appel système **fork ()** qui va dupliquer le processus appelant. A la suite de cet appel, les deux processus identiques continuent à s'exécuter à la suite du *fork ()*. La seule chose qui les différencient à ce moment est leur identifiant unique, ou encore **PID** (Process IDentifier). Ce numéro est renvoyé par l'appel système *fork ()*, comme déclaré da <*unistd.h*> :

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

C'est en fonction de ce code de retour que le processus sait si il est le père ou le fils. En effet, si on se trouve dans le processus **fils**, la valeur récupérée est **égale à zéro**.
On peut ainsi choisir d'exécuter deux codes différents à la suite du PID récupéré. Le processus fils pourra par exemple demander à être remplacé par le code d'un autre programme, c'est ce que fait le **shell** habituellement.
Un processus peut récupérer la valeur de son PID grâce à :

```
pid_t getpid(void) ;
```

Un processus fils peut, quand à lui accéder au PID de son père avec ;

```
pid_t getppid(void) ;
```

Voici un petit exemple des quelques fonctions que nous venons de voir :

EX1 *: Ecrire un programme qui affiche son PID dès son lancement, puis qui fork(). Le père et le fils devront respectivement renvoyer leur PID, le fils devra en plus renvoyer le PID de son père.*

> *Aide : Inclure l'appel* **wait(NULL) ;** *avant le retour du père, nous verrons son utilité plus tard.*

Observer le comportement du programme sans l'appel système ***wait() ;*** *et tenter de l'expliquer.*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

int main (void) {

     pid_t pid_fils;

     /*On affiche le PID du processus dès son lancement*/
     printf("Mon PID : %d\n", getpid());

     /*On exécute l'appel système fork()*/
     pid_fils = fork();

     /*Si on est le fils*/
     if (pid_fils == 0){
           printf("Mon PID (fils): %d\n", getpid());
           printf("Le PID de mon père: %d\n", getppid());
           return (0) ;
     }

     /*Si on est le père*/
     else{
           printf("Mon PID(père) : %d\n", getpid());
           wait(NULL);
           return (0) ;
     }

}
```

2.3.Identification de l'utilisateur correspondant au processus

2.3.1. Acquisition des informations relatives aux UID

Il est souvent indispensable de tirer partie de la gestion des utilisateurs sous Unix. Par exemple, un programme tel que ”login” (qui permet aux utilisateurs de se connecter à une console) vous permet, une fois authentifié par votre *nom d'utilisateur* et votre *mot de passe*, d'accéder au système tout en restant en accord avec vos droits.
Il existe trois niveaux d'identification pour les UID/GID :
Réel *: Il identifie qui vous êtes en réalité ; par exemple, si l'on travaille avec un UID différent du sien, l'UID réel reflète l'identité que vous possédez réellement.*
Effectif *: Il représente l'identification effective du processus. Il reflète le niveau d'accès dont dispose le processus sur le système.*
Sauvé *: Il permet de sauvegarder l'UID effectif du processus. Ainsi, lorsqu'un programme change d'UID (par exemple avec la fonction* **setuid***) il est toujours possible de revenir à l'UID initiale.*

Pour récupérer l'UID réel du processus en cours on utilisera la fonction :

```
uid_t getuid(void) ;
```

Pour récupérer l'UID effectif du processus en cours on utilisera la fonction :

```
uid_t geteuid(void) ;
```

Sous Unix, avec des fonctions comme ***setuid*** , les processus peuvent s'arranger pour obtenir de façon temporaire une valeur d'UID différente de celle qu'il possède au départ, d'où la distinction entre l'UID réel et l'UID effectif. Bien entendu, cette opération est soumise à de fortes restrictions par mesure de sécurité.

La façon la plus simple de mettre en évidence l'UID effectif et réel est de créer un petit programme qui affiche ces deux valeurs. Dans un premier temps celles-ci seront égales, par contre si nous activons le bit setuid la différence est flagrante :

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {


     /*On affiche l'UID réel du processus*/
     printf("UID réel : %d\n", getuid());


     /*On affiche l'UID réel du processus*/
     printf("UID effectif : %d\n", geteuid());

     return (0) ;

}
```

Si le programme ne possède pas le bit setuid root activé on obtient (en considérant l'UID de l'utilisateur appelant comme étant égal à 501):

```
UID réel : 501
UID effectif : 501
```

Activons maintenant le bit setuid root grâce à la commande chmod :

```
su –
password :
chown root.root notre_exemple
chmod +s notre exemple
ctrl-D   (pour se déloger)
```

On obtient alors à l'exécution (en tant qu'utilisateur) :

```
UID réel : 501
UID effectif : 0
```

A noter que 0 est l'UID de l'utilisateur root sur tout système GNU/Linux.

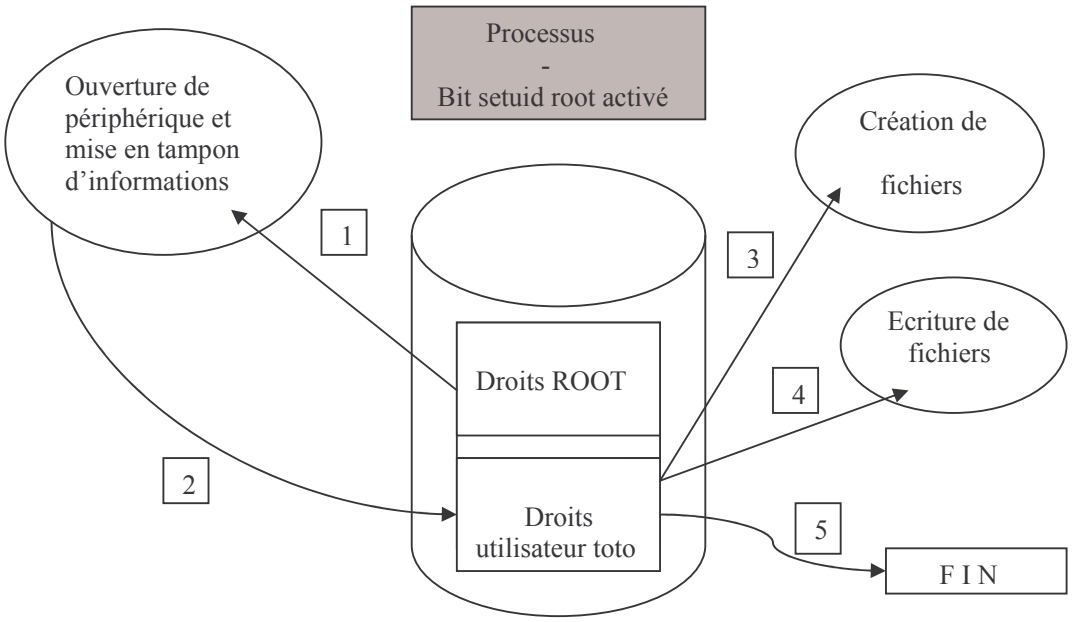
Une fonction nous permet de récupérer les trois types d'UID d'un seul coup, il s'agit de :

```
int getresuid(uid_t *ruid, uid_t *euid, gid_t *suid) ;
```

2.3.2. Changement d'UID

Maintenant que nous savons comment récupérer les informations relatives aux différents UID d'un processus voyons comment et dans quelle mesure nous pouvons les modifier.

Il existe plusieurs appels système permettant à un processus de modifier son UID. Il ne peut toutefois s'agir que de perdre ou éventuellement retrouver des anciens privilèges, mais jamais d'en gagner.
L'intérêt de ce mécanisme peut être résumé par le schéma suivant :



Ici notre processus a le bit setuid root activé, lors de son lancement l'UID effectif est donc celui de root. Ce qui permet à notre processus d'ouvrir un périphérique et de mettre en tampon les informations lues (1). Ensuite, pour des raisons de sécurité, il faut que celui-ci agisse avec les droits de l'utilisateur qui l'a invoqué (UID réel). Pour cela il perd ces droits (2) et effectue toutes les actions désirées (3) et (4) avant de se terminer.
La première fonction nous permettant de changer d'UID est

```
int setuid (uid_t uid_effectif) ;
```

Cette fonction est celle défini par la norme Posix.1. Attention si le programme est setuid root et qu'il est amené à perdre ces droits avec cette fonction, il sera incapable de les regagner !

L'autre fonction que nous allons voire et qui permet de récupérer n'importe quel type d'UID sauvé est :

```
int setreuid (uid_t uid_reel, uid_t uid_effectif) ;
```

Elle permet de définir les UID réel et effectifs d'un processus, si l'un des arguments vaut -1, alors la fonction ne les modifie pas.

Voici un exemple d'utilisation de la fonction setreuid() :

EX2 *: Ecrire un programme qui affiche son UID réel et effectif, puis qui remplace son UID effectif par son UID réel .Enfin après avoir affiché de nouveau les informations relatives à ses différents UID, le processus devra retrouver son UID effectif de départ.*
Observer le comportement du programme avec et sans le bit setuid root activé.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
     uid_t uid_reel ;
     uid_t uid effectif ;

     uid_reel = getuid() ;
     uid_effectif = geteuid() ;

     /*On affiche l'UID réel et effectif du processus*/
     printf("UID réel : %d  UID effectif %d \n", uid_reel
           , uid_effectif) ;

     /*On égalise l'uid effectif à l'uid réel*/
     setreuid(-1, uid_reel) ;

     /*On affiche l'UID réel et effectif du processus*/
     printf("UID réel : %d  UID effectif %d \n", uid_reel
           , uid_effectif) ;

     /*On récupère l'ancien uid effectif*/
     setreuid(-1, uid effectif) ;

     return (0) ;
```

2.4.Identification du groupe d'utilisateurs du processus

Chaque utilisateur du système appartient à un ou plusieurs groupes, ces derniers sont définis dans /etc/groups. Un processus fait donc également partie du ou des groupes de l'utilisateur qui l'a lancé.

Attention : Il ne faut pas confondre les **groupes d'utilisateurs** auxquels un processus appartient avec les **groupes de processus** qui permettent principalement d'envoyer des signaux à des ensembles de processus. Un processus appartient donc à deux types de groupes distincts qui n'ont rien à voire les uns avec les autres.

Comme pour les UID il existe 3 types de GID (Group IDentifier), un réel, un effectif et un sauvé. Mais çà la différence des UID, un processus peut avoir plusieurs GID (autant que le nombre de groupes auxquels appartient l'utilisateur). Le GID réel correspond au groupe principale de l'utilisateur.

Il est également possible d'attribuer à l'exécutable le drapeau **setgid** (chmod g+s), celui-ci a le même effet que le bit setuid mais se réfère aux groupes.

La lecture de ces GID se fait symétriquement à celle des UID avec les fonctions :

```
gid_t getgid (void) ;
gid_t getegid(void) ;
```

De même pour agir sur ces valeurs, on utilisera :

```
int setgid(gid_t egid) ;
int setregid(gid_t rgid, gid_t egid) ;
```

L'ensemble complet des groupes auxquels appartient un utilisateur est indiqué dans /etc/groups, un processus peut obtenir cette liste grâce à la fonction :

```
int getgroups (int taille, gid_t liste[]) ;
```

L'utilisation de cette fonction est un peu délicate, puisqu'il faut passer en premier argument la taille de la table qui va contenir le résultat. Cependant si on passe une taille nulle à cette fonction, elle nous renvoie le nombre de groupes supplémentaire du processus.

EX3 *: Ecrire un programme qui utilise correctement la fonction getgroups().*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>


int main (void) {
     int taille ;
     gid_t *table_gid = NULL ;
     int i ;

     /*On récupère la taille de notre table de gid*/
     if ( (taille = getgroups(0, NULL)) <0){
           fprintf(stderr, "Erreur dans getgroups : %d\n", errno) ;
           return (1) ;
     }

     /*On alloue de l'espace pour le tableau*/
     if ( (table_gid = calloc (taille, sizeof(gid_t))) == NULL ){
           fprintf(stderr, "Erreur dans calloc : %d\n", errno) ;
           return (1) ;
     }

     /*On Récupère la liste des groupes*/
     if (getgroups(taille, table_gid)){
           fprintf(stderr, "getgroups2 : %d\n", errno) ;
           return (1) ;
     }

     /*On affiche le résultat*/
     for (i=0 ;i < taille ; i++) {
           printf("%u", table_gid[i]) ;
     }
     printf("\n") ;
     free (table_gid) ;
     return (0) ;
}
```

2.5.Identification du groupe de processus

Les processus sont organisés en groupes. Ces groupes sont particulièrement utiles à l'envoi global de signaux à un ensemble de processus. Ceci est notamment utile aux interpréteurs de commande (shell) qui l'implémente pour le contrôle de jobs.
Pour savoir à quel groupe appartient un processus donné, on utilise la fonction :

```
pid_t getpgid (pid_t pid) ;
```

Elle prend en argument le PID d'un processus visé et renvoie son numéro de groupe ou -1 si le processus spécifié n'existe pas.

EX4 *: Ecrire un programme renvoie le numéro de groupe du processus en cours.*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

int main (void) {
     pid_t mon_pid ;

     mon_pid = getpid() ;
     printf ("PGID du processus : %u\n", getpgid(mon_pid)) ;

     return (0) ;
}
```

EX5 *: Même chose mais en spécifiant le PID du processus voulu en argument à la ligne de commande.*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

int main (void) {
     int i ;
     pid_t pid ;
     pid_t pgid ;

     /*Si aucun argument n'est passé le programme agis come l'ex4*/
     if (argc == 1){
            printf ("%u : %u\n" getpid(), getpgid(0)) ;
            return (0) ;
     }

     /*On parcours et traite la ligne de commande*/
     for (i=1 ; i< argc ; i++) {

            if (sscanf(argv[i], "%u", &pid) != 1){
                  fprintf(stderr, "PID invalide : %s\n",argv[i]) ;
             }
              else{
                  pgid = getpgid (pid) ;

                  if(pgid == -1) fprintf(stderr,"PID invalide\n") ;
                  else printf("%u : %u\n",pid,pgid) ;
       }
       return (0) ;
}
```

Lorsque plusieurs processus possèdent le même PGID, le processus qui a le même PID que le PGID en question est nommé leader du groupe. Notez cependant qu'un groupe n'a pas forcément de leader, celui-ci pouvant se terminer pendant que ses descendants continuent à s'exécuter.

Un processus peut modifier son propre identifiant de groupe ou celui de l'un de ses descendants avec la fonction :

```
setpgid(pid_t pid, pid_t pgid) ;
```

Le premier argument est le pid à modifier, le second le nouveau numéro de groupe. Si le premier argument est nul, l'action se fera sur le processus appelant.

Voici la procédure classique utilisée par les interpréteurs de commande :

- Faire un fork(), en gardant le résultat dans une variable.
- Le fils demande à devenir leader de son groupe (setpgid(0,0))
- Le père attend la fin de l'exécution du fils (waitpid())
- Le fils appelle une fonction du type exec() pour lancer la commande désirée.

Cette procédure assure au shell la possibilité de contrôler l'ensemble des processus appartenant au groupe du fils en leur envoyant des signaux.

3. Accès à l'environnement

Pour un processus, il peut s'avérer très utile de pouvoir accéder à son environnement. On entend par la non seulement à l'ensemble des variables d'environnement définies pour votre session mais également des arguments passés en ligne de commande. De plus vous pourrez être amené à devoir traiter de nombreuses options, certaines acceptant un argument, d'autre pas.
Il est évidement possible de mettre en place ces mécanismes d'accès à l'environnement externe au processus à la main, mais nous allons voir dans ce chapitre les différents moyens mis à notre disposition pour nous faciliter la tâche.

3.1.Accès aux arguments en ligne de commande

Considérez par exemple un programme de copie de fichier. Celui-ci peut demander de manière interactive le fichier source et destination de l'utilisateur, mais il est coutume de pouvoir transmettre ces informations directement après l'appel au programme, sur la ligne de commande :

```
cp fich1 fch2
```

Pour récupérer les valeurs des différents arguments, on déclarera notre fonction main() comme suit :

```
int main(int argc, char*argv[]){}
```

Une fois dans la fonction, **argc** représentera automatiquement le nombre d'arguments après le nom de l'exécutable invoqué + 1. En effet le nom du programme est considéré comme le premier argument.
De même **argv[i]** contient le i*ème* argument transmis (chaîne de caractère).
Voici un exemple simple qui utilise allègrement cette possibilité :
EX6: *Créer un programme agissant comme* ***echo*** *:*

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

/*Ce qu'il faut afficher si l'utilisateur passe moins de 1 arg*/
char *help = "Usage : ./mon_echo <arg1> ..\n";

int main (int argc, char *argv[])
{
     int i=1;

     if(argc < 2)
     {
            printf ("%s", help);
            return 1;
     }

/*Tant qu'il y a encore des args à traiter (--argc car il y a un
argument en trop dans argc : le nom du prog lui-même)*/

while (--argc > 0)
     {
           /*On affiche l'argument suivi d'un espace si ce n'est pas
           dernier argument ou d'un saut de ligne si c'est
           le dernier*/

            printf("%s%c", argv[i], (argc>1) ? ' ' : '\n');
            i++;
     }
return 0;
}
```

3.2.Traiter des options en ligne de commande

Vous pouvez grâce à ce que l'on viens de voire dans la section précédente parcourir, identifier, puis traiter par vous-même les éventuelles options présentes sur la ligne de commande. Celles-ci sont couramment utilisées dans le monde Unix, on trouve par exemple :
ls –li
rm –rf
ln –s
Cependant pour vous éviter un travaille fastidieux voici une méthode très efficace que vous pouvez employer sans modération ;

Il faut tout d'abord créer une chaîne de caractères contenant l'ensemble des fonctions que vous voulez traiter :

```
  char *liste_opts = "HhUN:PC";
```

Ici les options '-H', '-h', '-U','-P' et '-C' vont être reconnues comme option simple, le ':' suivant l'option '–N' précise qu'un argument (paramètre) sera passé à la suite de celle-ci.

Ensuite on entre dans une boucle en exécutant la fonction **getopt** à chaque tour jusqu'à ce que celle-ci renvoie -1, qui signifie que l'ensemble des arguments on été analysés. Avant d'avoir atteint la fin, getopt renvoie un entier correspondant à chaque 'lettre option' trouvée.
Il nous faut donc déterminer l'action à suivre pour chacune des lettre trouvées, ceci se fera aisément avec un switch (options) case … :

```
  [..]

  char *liste_opts = "hHUN:PCX:";

  [..]

  while( (option = getopt(argc, argv, liste_opts)) != -1)
       {
              switch (option)
              {
                     case 'h' :
                            printf("%s\n", help);
                            return 0;
                            break;

                     case 'H' :
                            Print_Env("Le   programme   s'execute   sur",
hostname, res);
                            break;

                     case 'U' :
                            Print_Env("L'utilisateur   appelant   est",
user, res);
                            break;

                     case 'P' :
                            Print_Env("Le     répertoire     perso     de
l'utilisateur est ",home, res);
                            break;

                     case 'C' :
                            Print_Env("Le      répertoire      courrant
est",cur_dir,res);
                            break;

                     case 'N' :
                            setenv("NEW",optarg,0);
```

```
                        break;
  [..]

  case '?' :
                        fprintf(stderr, "Option %c non prise en
charge, tapez ./td1 -h pour l'aide\n", optopt);
                        break;

            }
      }
```

Lorsque getopt tombe sur option inconnue, la fonction renvoie l'entier correspondant à '?'. Pour afficher notre propre message d'erreur au lieu de celui par défaut, on mettra à 0 la variable globale **opterr** au début du programme.

Enfin notez que pour une option à argument, on accède à son argument via la variable **optarg**.

3.3.Accès aux variables d'environnement

Les informations contenues dans les variables d'environnement peuvent s'avérer très utiles. On compte parmi elles le nom de la machine, de l'utilisateur en cours, du répertoire personnel de l'utilisateur, du répertoire courrant …
L'ensemble de ces variables sont accessibles dans votre programme C si vous déclarer avant votre main la variable externe **environ** comme suit :

```
  extern char **environ;
```

Ce tableau de chaînes de caractères est terminé par la valeur NULL. Pour afficher l'ensemble des variables globales on pourra donc écrire :

```
  while(environ[i] != NULL)
  {
       printf("%s\n",environ[i]);
       i++;
  }
```

Cependant chaque chaîne de caractère est composé d'un nom de variable suivi du signe égal suivi de la valeur associée. Pour nous éviter de séparer cela à la main et d'effectuer des recherches fastidieuses, on utilisera la fonction getenv() pour récupérer le valeur d'une variable d'environnement. Ainsi pour connaître afficher le nom de la machine et le répertoire courrant, on utilisera :

```
  printf("%s %s\n", getenv("HOSTNAME", "PWD"));
```

Vous pouvez être amené à modifier les valeurs d'une de vos variable durant l'exécution du programme. Ceci ce fait simplement avec la fonction **setenv()** ou **putenv**(). Voici le prototype de setenv :

```
  int setenv (char *name, const char *value, int overwrite);
```

En gros, cette fonction prend en premier argument le nom de la variable dont vous voulez définir la valeur située en second argument. Le troisième argument peut être 1 ou 0 et signifie respectivement que l'on peut ou non écraser la valeur de cette variable si elle existe déjà.
La fonction putenv reviens en fait à un setenv avec un troisième argument à un, mais la fonction ne prend qu'un argument composé de "NOM=VALEUR".

4. Exécution de programmes

4.1.La fonction popen()

La fonction popen() permet d'exécuter la commande passée en argument et retourne un flux unidirectionnel. On peut choisir de placer ce flux en lecture ou en écriture selon que l'on veut lire le résultat de la commande ou envoyer des données à la commande. Voici son prototype :

```
FILE *popen(const char *command, const char *type);
```

Lorsque vous avez fini d'utiliser le flux associé, il faut le refermer avec la commande **pclose()** et non fclose() :

```
int pclose(FILE *stream);
```

Voici un petit exemple qui affiche à l'écran le contenu du flux créé par popen() :

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

int main(void)
{
     FILE *cmd;
     char buffer[1024];
     if((cmd = popen("/bin/ps aux", "r")) == NULL)
            perror("popen");
     while(!feof(cmd))
     {
            fgets(buffer, 1023, cmd);
            puts(buffer);
     }

return 0;
}
```

Notez qu'il est très simple avec ce mécanisme d'écrire le résultat d'une commande dans un fichier.

4.2.La fonction system()

La fonction system() permet d'exécuter simplement tout type de commande prise en argument :

```
int system(const char *commande);
```

La valeur de retour de system est la valeur de retour du programme exécuté. Attention cependant, l'appel à system ne fait rien d'autre que d'exécuter /bin/sh –c <votre commande>. Ce qui implique que vous possédiez le Shell bash installé, c'est donc une fonction inutilisable pour la création d'un Shell par exemple.

4.3.La famille des exec()

Ces fonctions font toutes quasiment la même chose : lancer l'exécution d'un fichier *à la place du processus courant*. Elles diffèrent par la manière d'indiquer les paramètres. Contrairement à system() il faut bien comprendre que l'intégralité du processus appelant est remplacé par le code correspondant à la commande exécutée, c'est donc LA méthode à utiliser pour créer des applications indépendantes du Shell, voire même pour écrire un shell.

Voici les prototypes de tous les "execs" :

```
  #include <unistd.h>

  int execv (const char *FILENAME, char *const ARGV[])

  int execl (const char *FILENAME, const char *ARG0,...)

  int execve(const char *FILENAME, char *const ARGV[], char *const
ENV[])

  int execle(const char *FILENAME, const char *ARG0,...char *const
ENV[])

  int execvp(const char *FILENAME, char *const ARGV[])

  int execlp(const char *FILENAME, const char *ARG0, ...)
```

- execv() : les paramètres de la commande sont transmis sous forme d'un tableau de pointeurs sur des chaînes de caractères (le dernier étant NULL.
- *execl()* reçoit un nombre variable de paramètres, le dernier est NULL.
- *execve()* et *execle()* ont un param7tre supplémentaire pour préciser l'environnement.
- *execvp()* et *execlp()* utilisent la variable d'environnement **PATH** pour localiser l'executable à lancer. on pourrait donc écrire simplement :
- execlp("gcc","gcc",fichier,"-o",prefixe,NULL);

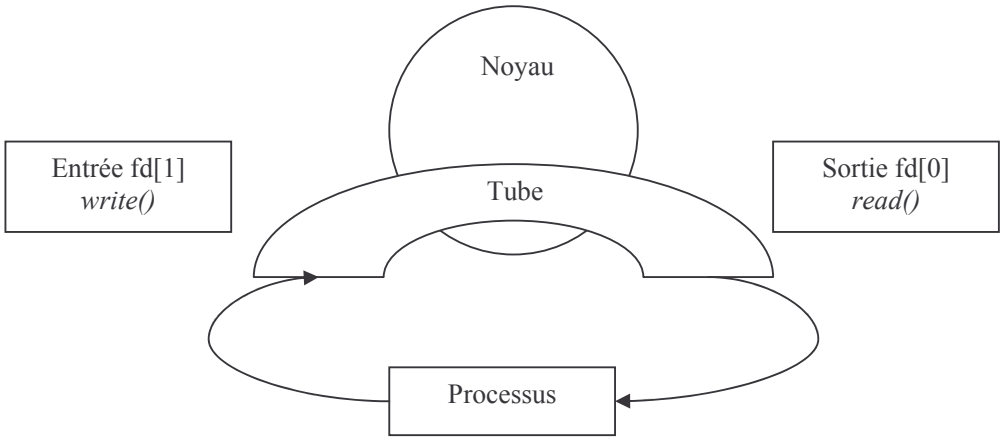
5. Communication classique entre père et fils

5.1.Tubes

Un tube de communication est un tuyau dans lequel un processus écrit des données qu'un autre processus peut lire. Ce tube est créé par l'appel à pipe() dont le prototype est :

```
int pipe (int fd[2]);
```

Lorsque cet appel réussi, il crée un nouveau tube au sein du noyau et rempli le tableau passé en argument avec les descripteurs des deux extrémités. A partir de ce moment fd[0] est la sortie du tube et fd[1] son entrée :



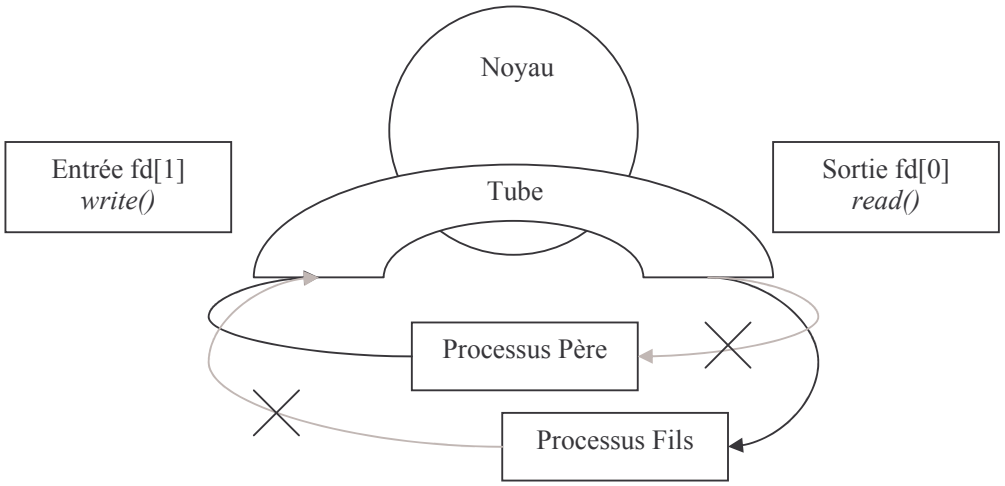
Comme le montre ce schéma un tube crée une communication unidirectionnelle, les deux descripteurs sont respectivement crées en lecture seule et en écriture seule, il faudra donc employer deux tubes dans les sens opposés.

Utiliser un tube au sein d'un même processus n'à que très peu d'intérêt, voyons comment nous pouvons employer un tube pour faire communiquer un père avec son fils. Lorsque vous faite un appel à fork(), tous les descripteurs se voient dupliqués, il convient donc de créer le tube avant d'appeler fork(). Ensuite il faut prendre conscience de cette règle :

Un appel comme read write ne débloquera que si :

- Plus aucune donnée n'est disponible dans le tube
- L'extrémité opposée est fermée.

Pour éviter des appels à read ou write bloquants indéfiniment, il convient donc de fermer dans le père et dans le fils les descripteurs qui ne nous intéressent pas :



Dans ce cas de figure le processus père écrit des données sur fd[1] que le fils pourra lire sur fd[0].
Nous allons maintenant voire comment dupliquer des descripteurs standards afin de les rattacher à l'une de nos extrémités de tube.

5.1.1. Duplication de descripteurs avec dup et dup2

L'appel dup fourni par le noyau permet d'effectuer permet d'obtenir une copie du descripteur fourni en argument :

```
dup(fd);
```

Le numéro associé dans la table des descripteurs du noyau sera le premier numéro de libre dans cette table.
Ce mécanisme est souvent utilisé pour rediriger les entrées et les sorties standard dans des tubes ou dans des fichier.

Voici la marche à suivre pour rediriger la sortie standard dans un fichier ouvert au préalable :

- Ouverture avec open du fichier
- Fermeture de la sortie standard close(STDOUT_FILENO). A ce moment le premier numéro disponible dans la table est donc 0.
- Duplication du descripteur correspondant à notre fichier, notre fichier se voit donc rattaché au descripteur 0 (sortie standard).
- Tout programme exécuté par la suite avec exec par exemple écrira donc dans notre fichier au lieu de l'écran.

L'appel dup possède cependant un point faible, si un autre processus ouvre un descripteur entre le moment où nous avons fermé stdout et le moment ou nous avons dupliqué notre descripteur de fichier, c'est le processus en question qui va ce voire attaché au descripteur 0.

Pour éviter ce type de problème le noyau met à notre disposition l'appel dup2(), qui lui effectue la redirection de manière atomique. Ainsi :

```
dup2 (fd, ancien);
```

permet de fermer le descripteur ancien si il est ouvert puis de dupliquer fd en lui associant une nouvelle entrée à la position ancien de la table des descripteurs.

Vous êtes maintenant en mesure de reprogrammer le '|' utilisé par le Shell !!

5.2.Tubes nommés

Le mécanisme de tube que nous venons de voire est très efficace, cependant il possède ses limite. Le tube étant crée dans le noyau, il n'est disponible que pour tous les processus descendant d'un ancêtre commun ayant créé le tube. On peut pourtant avoir besoin de faire communiquer deux processus indépendants, comme un serveur et un client par exemple. Pour cela il faut utiliser une extension des tubes vus précédemment : les tubes nommés.

Un tube nommé n'est rien d'autre qu'un nœud dans le système de fichier. Lorsqu'on l'ouvre pour la première fois, le noyau crée un tube de communication en mémoire. Ensuite les différents processus indépendants vont pouvoir écrire/lire depuis ce tube comme dans un tube simple vu auparavant. Les données contenues dans ce nœud sont du type FIFO (First In First Out).

La création d'un tel nœud ce fait avec :

```
int mkfifo(const char *nom, mode_t mode);
```

La fonction renvoie 0 en cas de réussite, le troisième argument de cette fonction est le même que pour la fonction open().

Il est possible, si vous préférez manipuler des flux, utiliser le jeu de fonctions fopen, fread,fwrite .. sur un tube nommé. Il est cependant préférable de l'ouvrir avec open() puis d'invoquer :

```
FILE *fdopen(int fd);
```

qui renvoie un flux correspondant au descripteur passé en argument.
Pour retirer un tube nommé du système de fichier on utilisera :

```
unlink(const char *nom_tube)
```

6. Un programme qui récapitule tout ..

Enfin Voici le listing d'un programme qui regroupe un bon nombre de notions vues et qui traite en plus des signaux classiques.

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <sys/types.h>
  #include <signal.h>
  #include <unistd.h>
  #include <assert.h>

  /*Fonction Macro qui essaye d'afficher le contenu d'une variable
d'environnement*/

  #define Print_Env(dsp,var_name, res)\
       res = getenv(var_name);\
       if (res == NULL) \
       {\
              fprintf(stderr, "La variable %s n'est pas définie\n",
var_name);\
       }\
       else\
              printf("%s %s\n", dsp, res);


  /*On récupère l'environnement*/
  extern char **environ;

  char *help ="-h : affiche cette aide\n-H : affiche le hostname\n-U :
affiche le User appelant\n-P : affiche le \
  répertoire perso de l'utilisateur\n-N <arg> défini une nouvelle Var
à exporter\n";

  void sortie_perso(void);
  void gestionnaire(int sig_num);

  int main (int argc, char *argv[])
  {
       /*Liste des options à traiter*/
       char *liste_opts = "hHUN:PCX:";
       int option;

       /*Pas de message d'erreur automatique*/
       opterr = 0;

       char *res;
       char *hostname = "HOSTNAME";
       char *user = "USER";
       char *home = "HOME";
       char *cur_dir = "PWD";
       char *fake = "CHOUBA";

       char *buf;
       char *message;

       FILE *sortie;

       pid_t pid;
       int int_ret,i=0;

       int fd[2];

       /*mise en place de la routeine de sortie personnalisée*/
       int_ret=atexit(sortie_perso);
```

```
	assert(int_ret==0);

	/*Mise en place du gestionnaire de signaux*/
	for(i=1;i<NSIG;i++)
	{
		if(signal(i, gestionnaire) == SIG_ERR)
		{
			fprintf(stderr, "Le signal %d ne peut être capturé
!\n", i);
		}
	}
	buf = (char *) calloc(1024,sizeof(char));
	assert(buf!=NULL);

	message = (char *)malloc(1024*sizeof(char));
	assert(message!=NULL);

	bzero((char *)buf, 1024*sizeof(char));

	bzero((char *)message, 1024*sizeof(char));

	printf("\n\t---Infos activés par options :---\n\n");

	/*Traitement des options passées en ligne de commande*/
	while( (option = getopt(argc, argv, liste_opts)) != -1)
	{
		switch (option)
		{
			case 'h' :
				printf("%s\n", help);
				return 0;
				break;

			case 'H' :
				Print_Env("Le programme s'execute sur",
hostname, res);
				break;

			case 'U' :
				Print_Env("L'utilisateur appelant est",
user, res);
				break;

			case 'P' :
				Print_Env("Le répertoire perso de
l'utilisateur est ",home, res);
				break;

			case 'C' :
				Print_Env("Le répertoire courrant
est",cur_dir,res);
				break;

			case 'N' :
				setenv("NEW",optarg,0);
				break;

			case 'X' :
				printf("\n>>>Commande exécutée par
l'utilisateur : %s\n", optarg);
				sortie = popen(optarg,"r");
				/*if( (sortie = popen(optarg,"r")) == NULL)
				{
					fprintf(stderr,"Erreur
d'execution\n");
					return 1;
				}*/
				assert(sortie != NULL);
```

```
                        while(!feof(sortie))
                        {
                              fread(buf,     sizeof(char),     1024,
sortie);
                              fwrite(buf,    sizeof(char),    1024,
stdout);

                        }

                        bzero((char *)buf, 1024*sizeof(char));

                        pclose(sortie);
                        break;
                  case '?' :
                        fprintf(stderr,  "Option  %c  non  prise  en
charge, tapez ./td1 -h pour l'aide\n", optopt);
                        break;

            }
      }


      printf("\n\t-------------------------\n\n");
      putenv("NEW2=ancient_value");
      setenv("NEW2","new_value2",0);
      Print_Env("La variable qui n'esiste pas est ",fake,res);
      putenv("CHOUBA=hello");
      Print_Env("La variable CHOUBA est maintenants ",fake,res);
      Print_Env("La variable NEW vaut ","NEW",res);
      Print_Env("La variable NEW2 vaut ","NEW2",res);
      printf("\n\t-------------------------\n\n");

      /*Création du pipe*/
      if(pipe(fd)!=0)
      {
            fprintf(stderr, "Probleme de Pipe\n");
            exit (1);
      }

      /*On lit dans fd[0] ce que l'on à écrit dans fd[1]*/

      /*Fork*/
      pid = fork();

      /*Si on est dans le fils*/
      if(pid == 0)
      {
            close(fd[0]);

            printf("Je suis le fils\n");
            printf("Mon UID réel est : %d\n", getuid());
            printf("Mon UID effectif est : %d\n", geteuid());
            printf("Mon GID réel est : %d\n", getgid());
            printf("Mon GID effectif est : %d\n", getegid());

            printf("Voici mon PID : %d\n", getpid());
            printf("Voici le PID de mon père: %d\n\n", getppid());

            printf("Entrez le message à envoyer au père : \n");

            fgets(message, 1024, stdin);

            write(fd[1], message, strlen(message));

            return 0;
      }

      /*Sinon on est dans le père*/
```

```
      else
      {
            close(fd[1]);
            wait(NULL);

            printf("\nJe suis le père\n");
            printf("Mon UID réel est : %d\n", getuid());
            printf("Mon UID effectif est : %d\n", geteuid());
            printf("Mon GID réel est : %d\n", getgid());
            printf("Mon GID effectif est : %d\n", getegid());

            printf("Voici mon PID : %d\n", getpid());
            printf("Voici le PID de mon père: %d\n\n", getppid());

            printf("Voilà le message que j'ai recu du fils (par le
pipe):\n");
            while(read(fd[0], buf, 1024))
            {
                  printf("%s", buf);
                  bzero((char *)buf, 1024*sizeof(char));
            }
            return 0;
      }
  }

  void sortie_perso(void)
  {
      printf("Aurevoire !!!\n");
      exit(0);
  }

  void gestionnaire(int sig_num)
  {
      fprintf(stdout, "Le signal %d à été receptionné <=> %s\n",
sig_num,sys_siglist[sig_num]);

  }
```