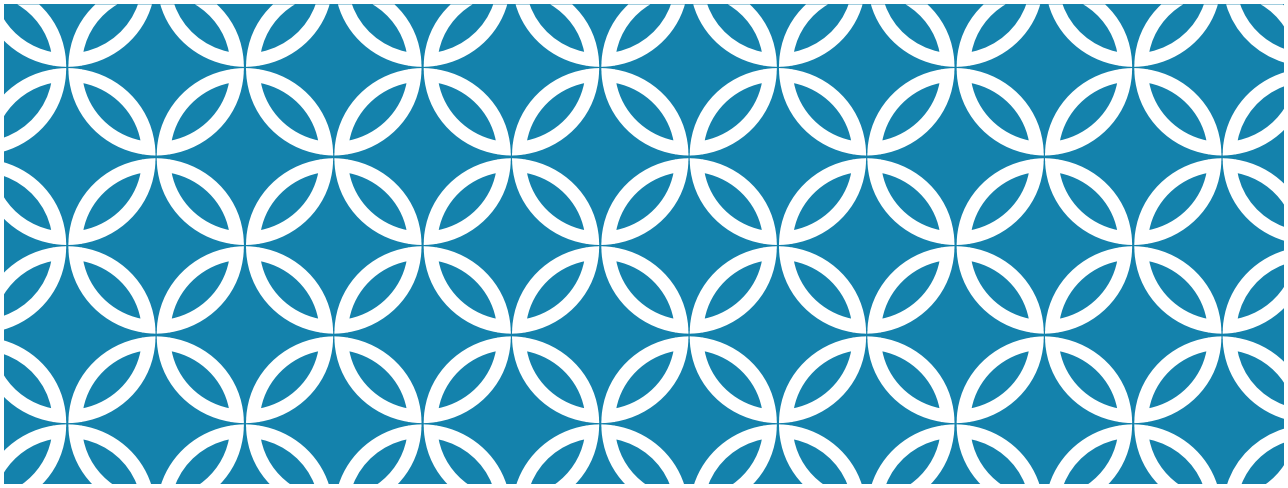


PROGRAMMATION SYSTÈME

Inter Connexion des Processus (IPC)

Mohamed Hammouda

LES TUBES : PIPE

- Un **tube** est un dispositif de communication qui permet une communication à sens unique.
- Un tube une fois initialisé admet deux extrémités :
  - **extrémité d'écriture**
  - **extrémité de lecture** .
- Les tubes sont des dispositifs séquentiels; les données sont toujours lues dans l'ordre où elles ont été écrites.
- Typiquement, un tube est utilisé pour la communication entre deux threads d'un même processus ou entre processus **père** et **fils**.
- Le **shell** crée également un tube connectant la sortie standard du processus ***ps*** avec l'entrée standard de ***grep***.
- Le résultat ***ps*** est envoyés à ***grep*** dans le même ordre que s'ils étaient envoyés directement au terminal.

```
% ps aux | grep atom
```

CRÉER DES TUBES

- La fonction ***pipe ( )*** *permet de créer un tube*.
- Pipe admet comme paramètre un **tableau de deux entiers**.
- Chaque entier, appelé descripteur de fichier, est un pointeur vers un fichier.
- La fonction renvoie **0** si elle réussit, et elle crée alors un nouveau tube.
- La fonction pipe remplit le tableau descripteur passé en paramètre, avec :
  - **pipe_fds [1]** désigne la sortie du tube (dans laquelle on peut lire des données) ;
  - **pipe_fds [0]** désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

```
int  pipe_fds [2] ;

pipe (pipe_fds);
```

CRÉER DES TUBES

- Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0].

- Le problème est qu'on ne crée le tube que dans **un seul processus**, et un autre processus ne peut pas deviner les valeurs du tableau descripteur.

- Pour faire communiquer plusieurs processus entre eux, il faut appeler la fonction **pipe** avant d'appeler la fonction **fork**.
- Le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux.

Pour écrire dans un tube, on utilise la fonction write :
size_t write (int descripteur1, const void *bloc, size_t taille);

Pour lire dans un tube, on utilise la fonction read :
size_t read (int descripteur0, void *bloc, size_t taille);

CRÉATION D'UN TUBE DANS UN PROCESSUS UNIQUE

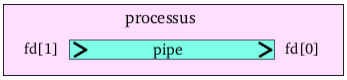
```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
```

```
int main( int argc, char ** argv ){
   char buffer[BUFSIZ+1];

   /* create the pipe */
   int fd[2];
   pipe(fd);

   /* write into the pipe */
   write(fd[1], "Hello World\n", strlen("Hello World\n"));

   /* read the pipe and print the read value */
   read(fd[0], buffer, BUFSIZ);
   printf("%s", buffer);
  }
```



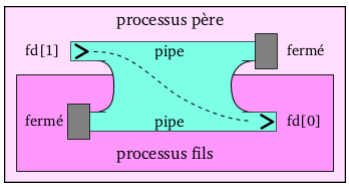
Créer un pipe dans un processus unique n'a pas beaucoup d'intérêt mais cela nous permet de comprendre ce qui caractérise un pipe :

- Un pipe possède deux extrémités.
- Il n'est possible de faire passer des informations que dans un sens unique.
- On peut donc écrire des informations à l'entré et en lire à la sortie.
- Les deux extrémités sont référencés par des descripteurs de fichiers (des entiers stockés dans la variable **fd**).

CRÉATION D'UN PIPE DANS UN PROCESSUS AYANT UN FILS

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

```
int main( int argc, char ** argv ){
  int  pfd[2];
  if (pipe(pfd) == -1)
    { printf("échec de création de pipe\n"); return 1; }
  int  pid;
  if ((pid = fork()) < 0) {
      printf("echec de création d'un fils\n");   return 2;   }
```



Si nous somme dans le processus Fils

```
if (pid == 0){
   char buffer[BUFSIZ];
   close(pfd[1]); /* close write side */
   /* read some data and print the result on screen */
    while (read(pfd[0], buffer, BUFSIZ) != 0)
        printf("child reads %s", buffer);
    close(pfd[0]); /* close the pipe */
}
```

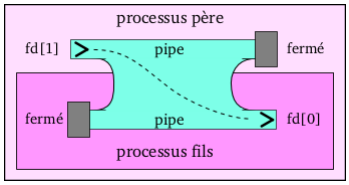
Si nous somme dans le processus Père

```
else {
   char buffer[BUFSIZ];
   close(pfd[0]); /* close read side */
    /* send some data into the pipe */
      strcpy(buffer, "HelloWorld\n");
      write(pfd[1], buffer, strlen(buffer)+1);
       close(pfd[1]); /* close the pipe */}
return 0;}
```

CRÉATION D'UN PIPE DANS UN PROCESSUS AYANT UN FILS

Écrire un programme qui crée deux processus.

- Le processus père ouvre un fichier texte en lecture. On suppose que le fichier est composé de mots formés de caractères alphabétiques séparés par des espaces.
- Le processus fils saisit un mot au clavier.
- Le processus père recherche le mot dans le fichier, et transmet au fils la valeur 1 si le mot est dans le fichier, et 0 sinon.
- Le fils affiche le résultat.



REDIRIGER LES FLUX D'E/S, D'ERREUR ET STANDARDS

- On peut lier la sortie tube[0] du tube à stdin. Par la suite, tout ce qui sort du tube arrive sur le flot d'entrée standard stdin, et peut être lu avec scanf, fgets, etc... Pour celà, il suffit de mettre l'instruction :

```
dup2(tube[0], STDIN_FILENO);
```

- De même, on peut lier l'entrée tube[1] du tube à stdout. Par la suite, tout ce qui sort sur le flot de sortie standard stdout entre ans le tube, et on peut écrire dans le tube avec printf, puts, etc... Pour celà, il suffit de mettre l'instruction :

```
dup2(tube[1], STDOUT_FILENO);
```

Plus généralement, la fonction dup2 copie le descripteur de fichier passé en premier argument dans le descripteur passé en deuxième argument.

REDIRIGER LES FLUX D'E/S, D'ERREUR ET STANDARDS

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

```
int main (){
   int fds[2];
  pid_t pid;
  pipe (fds);
  pid = fork ();
```

Si nous somme dans le processus Fils

```
 if (pid == (pid_t) 0) {
    close (fds[1]);
       /* Connexion de l'extrémité en lecture
        à l'entrée standard. */
       dup2 (fds[0], STDIN_FILENO);
       /* Remplace le processus fils par
       le programme "sort". */
       execlp ("sort", "sort", 0);
}
```

Si nous somme dans le processus Père

```
else {
  FILE* stream;
  close (fds[0]);
  stream = fopen (fds[1], "w");
  fprintf (stream, "C'est un test.\n");
  fprintf (stream, "Un poisson, deux poissons.\n");
  fflush (stream);
  close (fds[1]);
  /* Attend la fin du processus fils. */
  waitpid (pid, NULL, 0);
  }
return 0;
}
```

LES TUBES NOMMÉS

- On peut faire communiquer deux processus à travers un tube nommé. Le tube nommé passe par un fichier sur le disque. L'intérêt est que **les deux processus n'ont pas besoin d'avoir un lien de parenté**.
- Pour créer un tube nommé, on utilise la fonction **mkfifo** de la bibliothèque sys/stat.h.

- Une file *premier entré, premier sorti* (first-in, first-out, FIFO) est un tube qui dispose d'un nom dans le système de fichiers.
- Tout processus peut ouvrir ou fermer la FIFO; les processus raccordés aux extrémités du tube n'ont pas à avoir de lien de parenté. Les FIFO sont également appelés canaux nommés.

TUBES NOMMÉS : EXEMPLE

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

La fonction mkfifo prend en paramètre, outre le chemin vers le fichier, le masque des permissions (lecture, écriture) sur la structure *fifo*.

```
int main(){
 int fd;
 FILE *fp;
 char *nomfich="/tmp/test.txt"; /* nom du fichier */
 if (mkfifo(nomfich, 0644) != 0) /* création du fichier */
 {
   perror("Problème de création du noeud de tube");
   exit(1);
 }
 fd = open(nomfich, O_WRONLY); /* ouverture en écriture */
, "w"); /* ouverture du flot */
 fprintf(fp, "fp=fdopen(fdcoucou\n"); /* écriture dans le flot */
 unlink(nomfich); /* fermeture du tube */
 return 0;
}
```

le code suivant, le premier programme transmet le mot “coucou” au deuxième programme. Les deux programmes n’ont pas besoin d’être liés par un lien de parenté.

```
int main()
{
  int fd;
  FILE *fp;
  char *nomfich="/tmp/test.txt", chaine[50];
  fd = open(nomfich, O_RDONLY); /* ouverture du tube */
  fp=fdopen(fd, "r"); /* ouverture du flot */
  fscanf(fp, "%s", chaine); /* lecture dans le flot */
  puts(chaine); /* affichage */
  unlink(nomfich); /* fermeture du flot */
  return 0;
}
```

LES SIGNAUX

- Un signal permet de prévenir un processus qu’un évènement particulier c’est produit dans le système ou bien dans un **autre processus**.

- Plusieurs techniques sont utilisées pour envoyer des signaux :
  - par le noyau (comme en cas d’erreur de violation mémoire ou division par 0),
  - un programme utilisateur peut envoyer un signal avec la fonction ou la commande kill,
  - encore par certaines combinaison de touches au clavier (comme Ctrl-C).
  - Un utilisateur (à l’exception de root) ne peut envoyer un signal qu’à un processus dont il est propriétaire.

LES SIGNAUX

- Les principaux signaux (décrits dans la norme POSIX.1-1990) (faire man 7 signal pour des compléments) sont expliqués sur le table 7.1.

SIGHUP	Terminaison du leader de session (exemple : terminaison du terminal qui a lancé le programme)
SIGINT	Interruption au clavier (par Ctrl-C par défaut)
SIGQUIT	Quit par frappe au clavier (par Ctr − AltGr − \ par défaut)
SIGFPE	Exception de calcul flottant (division par 0 racine carrées d'un nombre négatif, etc...)
SIGKILL	Processus tué (kill)
SIGSEGV	Violation mémoire. Le comportement par défaut termine le processus sur une erreur de segmentation
SIGPIPE	Erreur de tube : tentative d'écrire dans un tube qui n'a pas de sortie
SIGTERM	Signal de terminaison
SIGSTOP	Stoppe temporairement le processus. Le processuss fige jusqu'à recevoir un signal SIGCONT
SIGCONT	Reprend l'exécution d'un processus stoppé.
SIGUSR1	**Réservé à l'usage par l'application**

ENVOYER UN SIGNAL

- La méthode la plus générale pour envoyer un signal est d'utiliser soit la commande shell kill(1), soit la fonction C kill(2).

La commande kill
La commande kill prend une option -**signal** et un **pid**.

Exemple.
$ kill -SIGINT 14764 {interromp processus de pid 14764}
$ kill -SIGSTOP 22765 {stoppe temporairement le process 22765}
$ kill -SIGCONT 22765 { reprend l'exécution du prcessus 22765}

La fonction kill

- La fonction C kill est similaire à la commande kill du *shell*. Elle a pour prototype

```
int kill (pid_t pid, int signal);
```

ENVOYER UN SIGNAL : EXEMPLE

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

Exemple. Le programme suivant tue le processus dont le *PID* est passé en argument seulement si l'utilisateur confirme.

```
int main (int argc, char ** argv)
{
  pid_t pidToSend;
  char rep;
  if (argc != 2)  {
    fprintf(stderr, "Usage %s pid\n", argv[0]);
    exit(1);
   }
   pidToSend = atoi(argv[1]);
   printf("Etes-vous sûr de vouloir tuer le processus %d ? (o/n)", pidToSend);
   rep = getchar();
   if (rep == 'o')
   kill (pidToSend, SIGTERM);
   return 0;
}
```

LE GESTIONNAIRE DES SIGNAUX

- Un gestionnaire de signal (*signal handler*) permet de changer le comportement du processus lors de la réception du signal (par exemple, se terminer).

Définition du handler

Une méthode qui sera invoquée une fois le processus encours recevra un signal.

```
void signal_handler (int signal) {
  printf(" réception du signal, traitement specifique encours,,,  ")
}
```

S'inscrire au gestionnaire d'évènement en définissant la méthode à exécuter pour chaque signal reçu

```
int main(int argc, char **argv) {
  Signal (SIGTERM, signal_handler)
  int i=0;
  while(1)
  {
      printf("itération numéro %d\n" ,i++);
      sleep(1);
   }
   return 0;}
```

LE GESTIONNAIRE DES SIGNAUX :

Exemple de programme qui sauvegarde des données avant de se terminer lors d'une interruption par Ctrl-C dans le terminal. Le principe est de modifier le comportement lors de la réception du signal SIGINT.

```
int donnees [5];

void gestionnaire(int numero)
{
  FILE *fp;
  int i;
  if (numero == SIGINT)
  {
     printf("\nSignal d'interruption, sauvegarde...\n");
     fp = fopen("/tmp/sauve.txt", "w");
     for (i=0 ; i<5 ; i++)
     {
        fprintf(fp, "%d ", donnees[i]);
     }
     fclose(fp);
     printf("Sauvegarde terminée, terminaison du
                                        processus\n");
     exit(0);
  }
}
```

```
int main (void){
   int i;
   char continuer='o';
   if (signalSIGINT, gestionnaire) != 0){
      fprintf(stderr, "Erreur sigaction\(\backslash\)n");
      exit(1);
   }
   for (i=0 ; i<5 ; i++){
      printf("donnees[%d] = ", i);
      scanf("%d", &donnees[i]); getchar();
   }
   while (continuer == 'o'){
      puts("zzz...");
      sleep(3);
      for (i=0 ; i<5 ; i++)
         printf("donnees[%d] = %d ", i, donnees[i]);
         printf("\nVoules-vous continuer ? (o/n) ");
         continuer = getchar();
         getchar();}
}
```

LES SIGNAUX : APPLICATION

- Ecrire un programme qui crée un fils qui fait un calcul sans fin. Le processus père propose alors un menu :
  - Lorsque l'utilisateur appuie sur la touche **'0'**, le processus père demande à son fils d'exécuter une tache particulère,
  - Lorsque l'utilisateur appuie sur la touche **'1'**, le processus père endort son fils.
  - Lorsque l'utilisateur appuie sur la touche **'2'**, le processus père redémarre son fils.
  - Lorsque l'utilisateur appuie sur la touche **'3'**, le processus près tue son fils avant de se terminer.