INP TOULOUSE ENSEEIHT

Systèmes d'exploitation : Unix

1 Le système Unix : introduction

1.1 Les origines

Le système Unix a pour origine les laboratoires BELL en 69-70. Les concepteurs K. Thompson et D. Ritchie avaient pour objectif de développer un système d'exploitation offrant un environnement de développement de programmes en mode interactif sur petit calculateur (PDP-9 doté de 32K octets de mémoire centrale...). À cette époque, seuls les gros systèmes (MULTICS, MVS/CTSS) offraient cette possibilité.

Conçu par ses futurs utilisateurs et par une équipe réduite, Unix est certainement d'une part, l'une des premières réactions contre le développement de systèmes énormes (pour l'époque, plusieurs millions de lignes de codes pour des systèmes du type de l'OS-360 d'IBM) contenant toujours quelques erreurs, et d'autre part, une application réussie du célèbre principe « small is beautiful » (certains concepteurs actuels de systèmes auraient peut-être intérêt à se souvenir de la démarche de leurs prédécesseurs).

Unix a été l'un des premiers systèmes écrit en langage évolué et non pas en assembleur. Le langage C a assuré une bonne portabilité du système d'exploitation, celui-ci ayant pu être mis en œuvre sans trop d'efforts de portage sur quantité de machines différentes. Là encore, les concepteurs d'Unix ont eu un rôle de précurseurs dans le souci d'ouvrir leur logiciel à tous et ce, presque gratuitement pour un usage non commercial.

Le succès d'Unix tient aussi à son langage de commande *shell*. Il est le premier système doté d'un langage interprété de commande ressemblant fort à un langage de programmation évolué classique. Seuls les objets manipulés diffèrent : les opérandes des commandes sont des noms de fichiers ou de programmes la plupart du temps. Doté de structures de contrôle (conditionnelle, boucles), il permet de développer des programmes ou scripts à un niveau de « programmation dans le large ». Un script décrit un traitement comportant éventuellement des activités parallèles et résultant d'un assemblage de « composants programmes ».

1.2 L'évolution

En 1975, une version stable, numérotée 6, écrite en C (pour 90% du code), est proposée et se répand dans le milieu universitaire. L'histoire d'Unix se scinde ensuite en deux :

- d'une part le milieu universitaire adopte majoritairement Unix avec l'université de Berkeley comme maître d'œuvre. Une version Unix-Berkeley, étiquetée BSD 4.x, est développée avec notamment un travail de recherche important sur la gestion mémoire (virtuelle) et la communication (réseau) ;
- d'autre part, une version plus industrielle étiquetée Unix/System V.

Un format de binaire différent interdisait l'échange de binaires exécutables et imposait un portage des programmes au niveau source. Quelques différences existaient aussi dans les primitives du noyau et les commandes shell. Heureusement, les deux branches se sont parlées... La plupart des versions Unix développées par les constructeurs offrent une union la plus cohérente possible des deux interfaces.

1.3 Le système Linux

Le système d'exploitation Unix a fait l'objet d'une normalisation appelée POSIX (Portable Operating System Interface) par l'IEEE (std 1003). Le système Linux (ou GNU/Linux) constitue une implantation de ce standard Unix. Le noyau de Linux a été développé à partir des années 90 par Linus THORVALDS. Celui-ci s'est inspiré de la démarche adoptée par Andrew TANENBAUM, professeur à l'université libre d'Amsterdam qui a implanté le premier clone Unix diffusé en version source (open source).

Combiné aux outils du projet GNU (« Gnu's Not Unix »), le succès de Linux est sans doute dû à la diffusion libre de ses sources, ce qui a permis un développement à la fois rapide et très conséquent de logiciels par une large communauté d'informaticiens internautes. Le projet GNU a conduit à l'émergence des logiciels libres et à la définition d'un type de licence appelée GNU GPL (General Public Licence) par Richard STALLMAN [1] et Eben MOGLEN.

Linux peut être déployé aujourd'hui sur de multiples configurations matérielles allant de processeurs embarqués aux super-calculateurs, son domaine de prédilection restant les ordinateurs personnels. Plusieurs diffuseurs de Linux se partagent le marché. On dispose ainsi de distributions RedHat, Debian, Suse, Ubuntu, etc. La distribution utilisée à l'N7 est diffusée par Ubuntu. Ces versions diffèrent essentiellement par leur facilité d'installation et de mises à jour du noyau, par les nombreux paquetages disponibles, par l'interface graphique usager, GNOME ou KDE, et par la richesse des pilotes de périphériques développés.

1.4 La conception générale du système

Une qualité sans doute essentielle d'Unix est de reposer sur un nombre restreint de concepts et principes. Pour le côté concept, le système Unix s'appuie sur deux concepts fondamentaux :

- pour la gestion des traitements (les exécutions de programmes), le concept de processus : ce concept encapsule l'exécution d'un programme de façon à assurer l'allocation et le contrôle des ressources nécessaires à l'exécution du programme. Par ailleurs, il permet d'optimiser l'utilisation des ressources concrètes ou abstraites disponibles (processeur(s), mémoires, périphériques, fichiers, pipes, sockets...) dans la mesure où plusieurs processus peuvent exister en parallèle ;
- pour la gestion des données, le concept de fichiers : ce concept permet d'assurer une rémanence et un contrôle des données des usagers du système.

Dans les deux cas, Unix gère un nombre dynamique d'objets processus ou fichiers et maintient une structure arborescente entre ces objets. En effet, tout processus est un descendant d'un processus unique initial racine et tout fichier est placé dans une arborescence de répertoires ayant une racine unique.

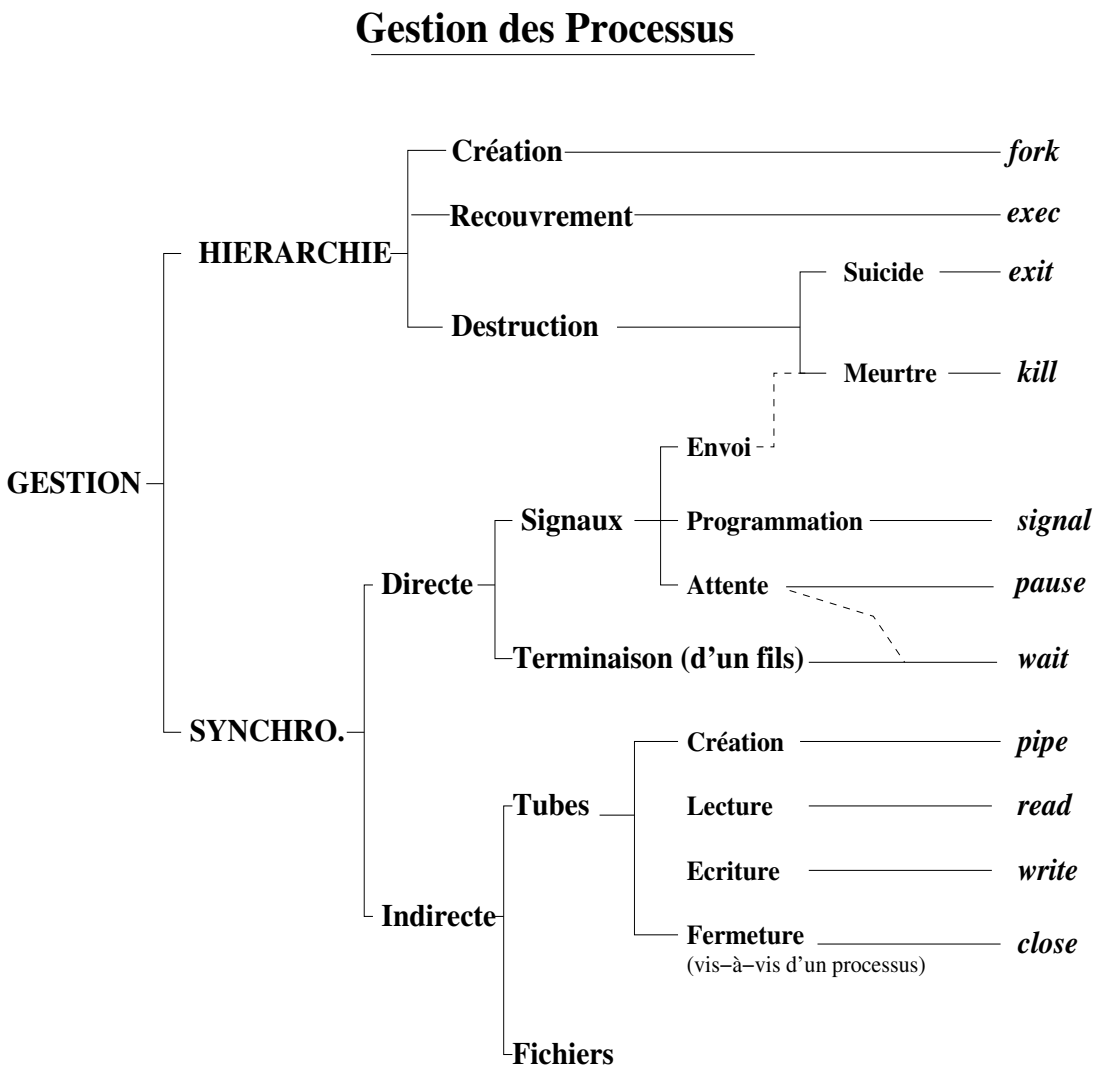
Processus et fichiers sont aussi réunis par la notion d'usager qui sert de base aux mécanismes de protection des ressources. Tout processus s'exécute pour un usager appartenant à un groupe. Tout fichier possède de la même manière un créateur identifié. L'identification d'un usager définit ainsi une capacité d'accès de cet usager aux ressources du système via les processus qu'il engendrera.

Parmi les principes, nous retiendrons :

- la transparence de l'architecture matérielle : ce principe de transparence consiste à masquer les particularités des architectures supports. Le noyau système définit par exemple un système de gestion d'exceptions (déroutements et interruptions), les signaux, indépendant de tout processeur matériel. Les périphériques peuvent être désignés symboliquement et sont intégrés dans l'arborescence de fichiers (répertoire `/dev`). Le concept de processus permet de masquer le nombre de processeurs réels...
- la création de nouveaux objets se fait souvent par duplication d'un objet existant, la nouvelle copie étant ensuite adaptée selon l'usage visé. Ceci évite des primitives à longues listes de paramètres (exemple typique, la création d'un processus se fait par une primitive `fork()` qui n'a **pas** de paramètre) et permet d'hériter implicitement de propriétés communes entre l'objet origine et sa copie.
- la destruction des objets rémanents tels que les fichiers, les pipes, est confiée au noyau. Celui-ci assure une fonction implicite de ramasse-miettes.

1. Président de la Free Software Foundation.

2 Gestion des Processus



2.1 Introduction

Les principaux outils nécessaires à la manipulation des processus sont résumés dans la figure suivante. Rappelons que les processus UNIX sont organisés de manière arborescente : à partir d'un processus initial *root*, tous les autres processus sont créés de « père en fils ».
Nous nous consacrons, dans un premier temps, aux aspects création, destruction, recouvrement, héritage d'un processus et synchronisation. Les primitives de gestion de processus `getpid, getppid, sleep, fork, exit, exec, kill, signal, pause` sont décrites ci-après. Par défaut et sauf spécifié autrement, les primitives sont déclarées dans `unistd.h`.

2.2 Les primitives de base

Les primitives `getpid, getppid, fork, exit, wait, exec` constituent les primitives de base de manipulation des processus. Leur syntaxe est brièvement décrite ci-après.

2.2.1 Identification de processus : `getpid` et `getppid`

```
pid_t getpid();
pid_t getppid();
```

Les fonctions à valeur entière `getpid()` et `getppid()` fournissent respectivement le numéro du processus appelant et celui du processus père (créateur). `pid_t` est un synonyme de `int` ou de `long`.

2.2.2 La fonction `sleep`

```
int sleep(int n);
```

`sleep` suspend l'exécution du processus appelant pour une durée de n secondes. Le processus est réveillé par l'envoi du signal `SIGALRM` au bout de n secondes.
Remarque : `sleep` utilise les primitives `alarm` et `pause` décrites ci-après, ainsi que le mécanisme des signaux.

2.2.3 La primitive `fork`

```
pid_t fork();
```

`fork()` permet la création dynamique d'un nouveau processus (dit processus *fils*) qui s'exécute de façon concurrente avec le processus qui le crée (dit processus *père*). Le processus fils hérite un certain nombre d'attributs du processus père. Un fils hérite :

- du même code,
- d'une copie de la zone de données du père,
- de l'environnement,
- de la priorité,
- des descripteurs de fichiers ouverts,
- du traitement des signaux.

La valeur de retour de `fork()` permet de distinguer facilement le processus fils créé (valeur 0) de son père (valeur égale au numéro du processus fils créé). C'est le seul moyen pour le père de connaître le numéro des processus qu'il crée. L'ordonnancement de l'exécution des processus père et fils est quelconque. Si la primitive `fork()` échoue alors sa valeur de retour est -1 (exemple : nombre maximal de processus atteint).

2.2.4 Processus, concurrence et ordonnancement

Exemple extrait de « *la programmation sous UNIX* » de Rifflet.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    int pid = fork(); /* père et fils */
    printf("Valeur de fork = %d ",pid);
    printf("processus %d de pere %d\n", (int) getpid(), (int) getppid());
    if ( pid == 0 ) { /* processus fils */ printf("fin du processus fils\n"); }
    else { /* processus pere */ printf("fin du processus pere\n"); }
    return 0;
}
```

Résultats de plusieurs exécutions :

```
prompt%creer_processus
Valeur de fork = 0 processus 434 de pere 433
fin du processus fils
Valeur de fork = 434 processus 433 de pere 364
fin du processus pere
prompt%

prompt%creer_processus
Valeur de fork = 0 processus 397 de pere 396
Valeur de fork = 397 processus 396 de pere 384
fin du processus pere
prompt%fin du processus fils

prompt%creer_processus
Valeur de fork = 440 processus 439 de pere 399
fin du processus pere
prompt%Valeur de fork = 0 processus 440 de pere 1
fin du processus fils
```

2.2.5 Exercices

1. On considère le programme suivant :

   ```
   #include <stdio.h>
   #include <unistd.h>
   #include <sys/types.h>
   int main(void)
   {
     fork(); printf("fork1\n");
     fork(); printf("fork2\n");
     fork(); printf("fork3\n");
     return(0);
   }
   ```

   Expliquez l'exécution de ce programme. Précisez en particulier le nombre total de processus engendrés par cette exécution. Combien d'occurrences de chaque type de messages `fork i` sont affichées ? Quel est l'ordre d'apparition des différents types de messages ?

2. Exécuter la commande `ps -Af` et reconstituer manuellement l'arbre de processus correspondant.

2.2.6 La primitive `exit`

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

`exit(n)` met fin au processus courant avec le code de retour n (par convention, le code de retour après un comportement correct est 0).

2.2.7 La primitive `wait`

```
#include <sys/wait.h>
pid_t wait(int *term);
```

`wait` provoque la suspension du processus appelant jusqu'à ce que l'un de ses processus fils se termine ou jusqu'à l'occurrence d'un signal émis vers le processus appelant. S'il n'existe aucun processus fils alors le retour de l'appel est immédiat et la valeur de retour est -1. Dans les autres cas, la valeur de retour est le numéro du processus fils qui a provoqué la fin de l'attente.
Si le processus fils s'est terminé normalement (par l'appel à la fonction exit) alors l'octet 0 du code de retour du processus fils est placé dans le deuxième octet de poids faible (octet 1) de l'entier pointé par `term`, les autres bits étant positionnés à 0.

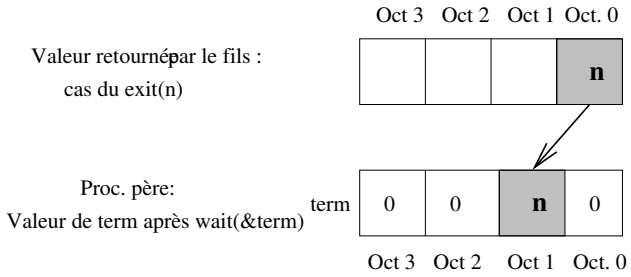


FIGURE 2.1 – Correspondances entre `exit` et `wait`

Si le fils ne s'est pas terminé normalement du fait de la réception d'un signal, le premier octet de poids faible (octet 0) de l'entier pointé par `term` contient le numéro du signal. Des macros (faire `man -s 2 wait`) permettent de traiter la valeur de retour `n` et de distinguer les différents cas de retour (`WIFSIGNALED`, `WIFEXITED`, `WTERMSIG`, `WEXITSTATUS`...).
Un processus fils qui se termine avant la fin du père devient un processus « zombie » jusqu'à l'exécution d'un `wait` ou jusqu'à la fin du processus père.
Remarques :

- si le père se termine sans prendre en compte la terminaison du fils, le fils est alors adopté par le processus initial 1 qui traitera sa terminaison ;
- un processus père peut attendre la fin de tous ses fils en effectuant autant d'appels à `wait` qu'il a de processus fils ;
- la primitive `waitpid` permet d'attendre la fin d'un processus fils dont le numéro est passé en paramètre.

2.2.8 La famille `exec`

Les primitives `exec` sont une famille de primitives permettant le lancement de l'exécution d'un nouveau programme par un processus appelant. Cela n'entraîne pas la création d'un nouveau processus mais seulement le chargement et le lancement d'un nouveau programme exécutable, en lieu et place du programme contenant l'appel à la primitive `exec` : le nouveau programme recouvre (remplace directement) l'ancien.
Les descripteurs de fichiers ouverts avant l'appel d'un `exec` le restent. Si on utilise des stream `FILE *` (`fopen, printf`...), alors il y a écrasement du tampon associé en zone utilisateur et donc risque de perte

d'information. Pour forcer le vidage de ce tampon avant l'appel à `exec` on utilise la fonction `fflush()` de la bibliothèque standard (inclure `stdio.h`).

```
int execl(char *ref, char *arg0, char *arg1, char *arg2,..., char *argn, NULL);
int execv(char *ref, char *argv[]);
```

`execl` et `execv` permettent le lancement de l'exécution du fichier exécutable dont le nom est `ref` avec les arguments `arg0, arg1, arg2,...,argn` pour `execl` et les différentes chaînes de caractères pointées par `argv` pour `execv`. Le premier paramètre (`arg0` ou `argv[0]`) est obligatoire et pointe en général sur une chaîne identique à celle pointée par `ref`. Le fichier donné en référence doit exister et correspondre à un fichier exécutable.

Variantes
`execlp` et `execvp` sont deux variantes de respectivement `execl` et `execv`, mais avec recherche du fichier exécutable dans la liste des répertoires contenue dans la variable `PATH`.
`execle` et `execve` sont deux variantes de respectivement `execl` et `execv`, prenant un argument supplémentaire (un tableau de chaînes de caractères) qui permet de spécifier la liste des variables d'environnement du nouveau programme.
En cas d'échec de l'appel à l'une des fonctions `exec`, la valeur de retour est -1.

Note : ces primitives donnent le contrôle à un programme nouveau sans espoir de retour à l'appelant (celui-ci étant écrasé) sauf, bien sûr, dans le cas où l'appel à `exec` échoue.

Exercice d'application : un processus lance en tâche de fond une recherche avec affichage des fichiers de suffixe « `.c` » présents à partir du répertoire courant. Aussitôt et en parallèle, il travaille 5 secondes (simulé à l'aide de `sleep`). À l'issue de ce travail, il attend (éventuellement) la fin de la tâche qu'il a lancée (on ne connait pas le temps de l'exécution de cette tâche).
Enfin, il vérifie la « qualité » de l'exécution de la tâche en analysant puis en affichant le compte-rendu complet de son exécution. Un message final achève son travail.

2.2.9 Exercice

Afin d'illustrer l'utilisation et les propriétés de ces primitives de base, écrire un programme qui d'une part illustre la propriété « une variable globale n'est pas partagée par un processus et son fils » et d'autre part effectue les actions suivantes :

- Créer un processus fils
- Dans le processus fils :
  - imprimer les valeurs retour de fork, getpid, getppid ;
  - bloquer 4 secondes le processus ;
  - le processus se termine par un appel à exit(2) ;
- Dans le processus père :
  - imprimer la valeur de fork, getpid, getppid ;
  - mettre en attente le processus de la fin de son fils ;
  - imprimer le code de retour du processus fils. Quelle valeur doit on obtenir ?
  - lancer l'exécution (sans création de nouveau processus), d'un programme écrit en C qui imprime le numéro de processus courant puis liste tous les processus actifs (commande `ps -Al`).
- (En travaux pratiques :)
  - tester le code ;
  - analyser les résultats obtenus ;
  - comparer les résultats d'une exécution du programme avec et sans redirection des sorties ;
  - expliquer la différence entre les deux résultats.

2.2.10 Exercice : gestion de la mémoire

Décrire précisément le résultat de l'exécution du programme ci-dessous (affichage, création de processus, filiation des processus...)

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

int v_glob = 4;

void f_with_static(int val)
{
  static int v_stat = 0;
  printf("debut f_with_static : v_stat=%d parametre=%d\n",v_stat,val);
  v_stat = val;
  printf("fin f_with_static : %d\n",v_stat);
}

int main(void)
{
  int pid, m, n, *p_int;

  p_int = (int *) malloc (sizeof(int));
  *p_int = 19;

  f_with_static(34);

  if ( (pid=fork()) == 0 ) { /* fils */
     printf("on est dans le fils %d de pere %d\n",(int) getpid(),(int) getppid());
     v_glob++;
     (*p_int)++;
     printf("fils: v_glob apres incrementation=%d\n",v_glob);
     printf("fils: donnee dynamique apres incrementation=%d\n",*p_int);
     printf("fils: appel de f_with_static avec %d\n",v_glob);
     f_with_static(v_glob);
     exit(2);
  }
  /* père */
  printf("on est dans le pere %d de pere %d\n",(int) getpid(),(int) getppid());
  m = wait(&n);
  printf("le fils est mort, retour au pere : m= %d n= %d\n",m,n);
  printf("pere: v_glob=%d\n",v_glob);
  printf("pere: donnee dynamique=%d\n",*p_int);
  printf("pere: appel de f_with_static avec %d\n",*p_int);
  f_with_static(*p_int);
  return(0);
}
```

2.3 Les primitives de synchronisation directe entre processus

Les signaux constituent un mécanisme fondamental de communication entre processus. Les signaux sont identifiés par un nombre entier. Le signal `SIGINT` (en général engendré par `CTRL-C`) et la commande `kill` sont couramment manipulés par l'utilisateur pour interrompre des processus.

2.3.1 Liste des signaux

Le fichier `/usr/include/signal.h` contient la liste des signaux accessibles sur un système donné. Les noms sont standards mais les valeurs numériques peuvent changer selon le système.

```
#define SIGHUP 1 /* Hangup (POSIX).  */
#define SIGINT 2 /* Interrupt (ANSI).  */
#define SIGQUIT 3 /* Quit (POSIX).  */
#define SIGILL 4 /* Illegal instruction (ANSI).  */
#define SIGTRAP 5 /* Trace trap (POSIX).  */
#define SIGABRT 6 /* Abort (ANSI).  */
#define SIGIOT 6 /* IOT trap (4.2 BSD).  */
#define SIGBUS 7 /* BUS error (4.2 BSD).  */
#define SIGFPE 8 /* Floating-point exception (ANSI).  */
#define SIGKILL 9 /* Kill, unblockable (POSIX).  */
#define SIGUSR1 10 /* User-defined signal 1 (POSIX).  */
#define SIGSEGV 11 /* Segmentation violation (ANSI).  */
#define SIGUSR2 12 /* User-defined signal 2 (POSIX).  */
#define SIGPIPE 13 /* Broken pipe (POSIX).  */
#define SIGALRM 14 /* Alarm clock (POSIX).  */
#define SIGTERM 15 /* Termination (ANSI).  */
#define SIGSTKFLT 16 /* Stack fault.  */
#define SIGCLD SIGCHLD /* Same as SIGCHLD (System V).  */
#define SIGCHLD 17 /* Child status has changed (POSIX).  */
#define SIGCONT 18 /* Continue (POSIX).  */
#define SIGSTOP 19 /* Stop, unblockable (POSIX).  */
#define SIGTSTP 20 /* Keyboard stop (POSIX).  */
#define SIGTTIN 21 /* Background read from tty (POSIX).  */
#define SIGTTOU 22 /* Background write to tty (POSIX).  */
#define SIGURG 23 /* Urgent condition on socket (4.2 BSD).  */
#define SIGXCPU 24 /* CPU limit exceeded (4.2 BSD).  */
#define SIGXFSZ 25 /* File size limit exceeded (4.2 BSD).  */
#define SIGVTALRM 26 /* Virtual alarm clock (4.2 BSD).  */
#define SIGPROF 27 /* Profiling alarm clock (4.2 BSD).  */
#define SIGWINCH 28 /* Window size change (4.3 BSD, Sun).  */
#define SIGPOLL SIGIO /* Pollable event occurred (System V).  */
#define SIGIO 29 /* I/O now possible (4.2 BSD).  */
#define SIGPWR 30 /* Power failure restart (System V).  */
#define SIGSYS 31 /* Bad system call.  */
```

Parmi la liste des signaux précédents, nous utiliserons principalement les signaux `SIGINT, SIGQUIT, SIGKILL, SIGPIPE, SIGCHLD, SIGUSR1, SIGUSR2` pour synchroniser et faire communiquer des processus.
À chaque signal est associé initialement un traitement par défaut. Il existe cinq types de traitements différents : la terminaison (`SIGINT, SIGQUIT`...), la terminaison avec création d'un fichier `core` (`SIGSEGV`), l'ignorance (`SIGCHLD` signal délivré automatiquement au père lors de la mort de tout processus fils), la suspension (`SIGSTOP`) et la reprise (`SIGCONT`).

2.3.2 Émission d'un signal : `kill`

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

`kill` permet l'émission du signal de numéro `sig` à destination du processus de numéro `pid`. Si `sig` est nul, aucun signal n'est envoyé et l'opération est réussie ssi le processus de numéro `pid` existe. La valeur de retour indique si l'opération est réussie ($retour = 0$) ou non ($retour = -1$).

2.3.3 Comportement lors de la réception d'un signal : `signal`

```
#include <signal.h>
typedef void (*handler_t)(int); /* procédure prenant un entier en paramètre */
handler_t signal (int sig, handler_t handler)
```

La plupart du temps, la réception d'un signal entraîne la terminaison du processus. Cependant, certains signaux tels que `SIGCHLD` (signal généré en particulier automatiquement et reçu par le père à la mort de chaque processus fils) et `SIGCONT` sont par défaut ignorés.
L'émission d'un signal peut être provoquée par :

- le système (erreur de programme, ou interruption système),
- une interruption de l'utilisateur depuis son terminal,
- `kill` effectué par un autre processus.

Le comportement d'un processus à la réception d'un signal peut cependant être modifié : il est possible d'associer un traitant (*handler*) particulier à la réception d'un signal. Pour cela, la fonction `signal` considère deux paramètres qui sont le numéro du signal et la fonction à exécuter si ce signal est reçu. De plus, `signal` renvoie la fonction qui était précédemment associée au signal.

```
void fonc (int sig)
{
   printf("numero signal %d\n", sig);
}
...
signal(sig, fonc);
...
```

Lorsque le signal `sig` est reçu :

- Le numéro de signal (`sig`) est alors fourni comme argument à la fonction de traitement (`fonc(sig)`) ;
- Après exécution de `fonc(sig)`, l'exécution du processus reprend au point où il avait été interrompu.
- Il existe deux sémantiques sur la réinitialisation du traitant d'un signal :
  - sémantique System V : Le processus n'est pas masqué contre le signal `sig` durant l'exécution de `fonc` et le traitement par défaut du signal est automatiquement réassocié au signal `sig` avant l'exécution du traitant `fonc`. Quel(s) problème(s) pose(nt) ce comportement ? comment y remédier ?
  - sémantique BSD : le signal `sig` est masqué durant l'exécution de `fonc` et le traitant installé est conservé après le déroutement.

Deux fonctions particulières sont prédéfinies dans le fichier `signal.h` :

`SIG_DFL` désigne le comportement par défaut associé à la réception du signal. Un signal a un comportement par défaut, prédéfini est associé à chaque signal : terminer le processus récepteur (le plus fréquent), ignorer le signal, suspendre le processus ... ;

`SIG_IGN` désigne le comportement consistant à ignorer le signal (traitant vide).

Vis-à-vis des signaux, un processus fils hérite du comportement de son père. Ce n'est pas le cas lors du recouvrement d'un processus (`exec`) où seuls le comportement par défaut (`SIG_DFL`) et le traitant ignorant le signal (`SIG_IGN`) peuvent être transmis à la nouvelle image.

2.3.4 Primitives de manipulation du masque des signaux : `sigprocmask, sigsetops, sigpending, sigsuspend, sigaction`

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

`sigprocmask` permet de consulter et d'affecter le masque des signaux du processus appelant. Ce masque définit l'ensemble des signaux qui sont masqués pour le processus. `how` peut prendre les trois valeurs suivantes :

- `SIG_BLOCK` : `set` est ajouté au masque courant ;
- `SIG_UNBLOCK` : `set` est retiré du masque courant ;
- `SIG_SETMASK` : `set` remplace le masque courant.

Si `oset` est différent de `NULL`, le masque précédent est placé à cette adresse.
Si `set` vaut `NULL`, `how` n'est pas significatif, le masque courant reste inchangé :
`sigprocmask(SIG_BLOCK, NULL, &masq_courant)` permet de récupérer le masque courant dans l'ensemble `masq_courant`.
Pendant un masquage, seule une occurrence de chaque signal masqué est mémorisée ; si le signal est démasqué, cette occurrence est délivrée.
`man sigsetops` fournit la description des fonctions manipulant les masques :

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Les deux premières fonctions sont à utiliser *avant* les trois suivantes afin d'initialiser le masque manipulé. Les identificateurs de fonctions sont suffisamment expressifs pour ne pas détailler leur sémantique.

```
#include <signal.h>
int sigpending(const sigset_t *set);
```

`sigpending` indique en retour dans `set` les signaux masqués pour lesquels un signal a été bloqué et non délivré (signaux pendants).

```
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

Cette fonction remplace le masque courant par le masque `set` et suspend le processus appelant jusqu'à l'arrivée d'un signal provoquant l'exécution d'un traitant ou l'arrêt du processus. Dans le cas de l'exécution d'un traitant, une fois cette exécution terminée, l'exécution du processus reprend après l'appel à `sigsuspend` après avoir restauré le masque initial.
Il existe également une fonction `sigaction` qui permet une manipulation complète des gestionnaires de signaux (cf. `man sigaction` pour plus de précisions).

2.3.5 Attente d'un signal : `pause`

```
int pause();
```

La fonction `pause` met le processus appelant en attente de l'arrivée d'un signal quelconque, à condition qu'il ne soit pas ignoré ou masqué par le processus. Le réveil du processus par un signal met par défaut fin à ce processus sauf si un traitant a été associé à ce signal. Dans un tel cas, `pause()` restitue la valeur -1 et le processus reprend après l'exécution du traitant.

2.3.6 La primitive `alarm`

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

`alarm` entraîne l'envoi du signal `SIGALRM` au processus appelant après un laps de temps `sec`. Des appels successifs à la primitive provoquent une réinitialisation de l'horloge à la dernière valeur spécifiée. La valeur de retour est le temps restant dans l'horloge. Après émission du signal l'horloge prend la valeur 0.
Remarque : Un appel à `alarm` peut être perturbé par d'autres primitives qui utilisent aussi `alarm` (par exemple `sleep`).

2.3.7 Programmation d'horloges : `setitimer, getitimer`

```
#include <sys/time.h>

int getitimer (int which, struct itimerval *value)
int setitimer (int which, struct itimerval *value, struct itimerval *ovalue)

/* Les structures suivantes sont definies dans <sys/time.h>. */
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
};
struct  itimerval {
        struct  timeval it_interval;    /* intervalle du timer */
        struct  timeval it_value;       /* valeur courante */
};
```

Les primitives systèmes `getitimer` et `setitimer` permettent de manipuler trois horloges associées à un processus. Chacune des trois horloges a un fonctionnement distinct. Le paramètre `which` spécifie l'horloge utilisée. À une horloge est associée une structure (voir `itimerval`) indiquant une valeur courante (`it_value`) et un intervalle (`it_interval`). À chaque type d'horloge est associé un signal (`SIGALRM`, `SIGVTALRM` ou `SIGPROF`) qui sera envoyé au processus au bout de `it_value` avec une période de `it_interval`.
Si le champ `it_value` d'une horloge est égal à zéro alors l'horloge est désactivée. Si `it_interval` est égal zéro alors l'horloge sera désactivée après la prochaine émission d'un signal de l'horloge (au bout du temps indiqué par `it_value`$\neq$`0`). La macro `timerclear()` définie dans `<sys/time.h>` permet de remettre à zéro une variable de type `timeval`.
`getitimer` permet de récupérer (dans la structure pointée par `value`) les caractéristiques de l'horloge spécifiée par `which`.
La primitive `setitimer` positionne l'horloge spécifiée par `which` à la valeur pointée par `value` et renvoie dans `ovalue` l'état précédent de l'horloge.
Valeurs possibles de l'horloge (`which`) :

- `ITIMER_REAL` : l'horloge est décrémentée en temps réel, `SIGALRM` est envoyé au processus à chaque fin de période.
- `ITIMER_VIRTUAL` : l'horloge est décrémentée uniquement lorsque le processus est actif (temps virtuel), `SIGVTALRM` est envoyé au processus à chaque fin de période.
- `ITIMER_PROF` : l'horloge est décrémentée lorsque le processus est actif ou lorsque le processus est dans le noyau (temps système). `SIGPROF` est envoyé au processus à chaque fin de période.

Il existe une quatrième horloge `ITIMER_REALPROF` associée également au signal `SIGPROF` et utilisée dans le cadre de programmes « multithread ».

2.4 Gestion des points de reprise

Les fonctions `setjmp, longjmp` sont utiles pour la mise en place et l'utilisation de points de reprise. Ces primitives permettent notamment la mise en œuvre du mécanisme d'exception ou l'implantation de processus légers. Ces notions seront développées ultérieurement.

```
#include <setjmp.h>
int setjmp(jmp_buf env)
void longjmp(jmp_buf env, int val);
```

`setjmp(env)` sauvegarde dans la variable `env` le contexte d'appel du processus courant (pc, registres...). La valeur de retour est égale à zéro pour un appel normal.
`longjmp(env, val)` permet de restaurer l'environnement décrit par le paramètre `env`. L'exécution du processus se continue alors comme si l'appel à `setjmp(env)` venait d'être effectué. La valeur de retour de `setjmp` est cependant dans ce cas égale à la valeur passée en paramètre de l'appel à `longjmp` (sauf lorsque $val = 0$ où `setjmp` renvoie 1).

```
#include <setjmp.h>
int val;
jmp_buf env;
...
/* sauvegarde d'un point de reprise */
val = setjmp(env);
if (val==0) {
   ...
   /* Cascade d'appels proceduraux  */
       ....
       /* Detection d'un probleme et retour au point de reprise */
       longjmp(env, 1);
   ...
} else {
   /* traitement apres longjmp  */
}
```

2.5 Exercices

1. Décrire précisément le résultat de l'exécution du programme ci-dessous (affichage, création de processus, filiation des processus...)

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

   int main(void) {
     int i, pid;
     i=0;
     pid=0;
     printf("je suis %d\n", (int) getpid());
     while ( (i < 3) && (pid == 0) ) {
        i++;
        pid = fork();
     }
     printf("pid=%d ppid=%d\n", pid,  (int) getppid());
     return 0;
   }
   ```

2. Écrire le programme suivant : un processus crée cinq processus fils. Chaque processus fils (de numero i) se termine par un exit(i). Le processus père crée également un sixième processus fils qui entre dans une boucle infinie et sera tué par l'utilisateur. Le processus père attend la fin de tous ses fils. Pour chaque fils il affichera un message lisible indiquant la cause de la mort du fils.
3. Écrire un code C qui imprime le numéro de tout signal reçu (signal émis depuis le terminal ou depuis un autre processus). Ce code indiquera toutes les 3 secondes qu'il est toujours actif et en attente de signaux. Au bout de 5 signaux recus ou 27 secondes il devra s'arrêter. `SIGALRM` est le signal utilisé par la fonction `alarm` mais peut aussi être engendré par tout processus à l'aide de `kill`. On ne devra compter que les signaux `SIGALRM` résultant d'appels à `kill`.
4. Écrire un code C tel que le processus correspondant ait le comportement suivant : lorsqu'il reçoit le signal `SIGINT` (engendré par la touche ctrl-C) depuis le terminal, il entre dans une boucle infinie que l'on peut interrompre seulement en lui envoyant un des signaux non masquables (par exemple le signal 9). Dans cette boucle, le processus imprimera par exemple son numéro toutes les cinq secondes. On traitera cet exercice avec ou sans masquage des signaux.
5. Ecrire un code qui illustre l'utilisation des primitives `signal, pause, wait, kill`. Plus précisément, on souhaite :
   — illustrer l'héritage des signaux dans un processus fils (primitive `fork`) ;
   — déterminer l'héritage des signaux lors du lancement de l'exécution d'un nouveau programme en utilisant la famille de primitives `exec`.

   On considèrera trois types de comportement vis-à-vis de la réception d'un signal : le comportement par défaut (`SIG_DFL`), l'ignorance du signal (`SIG_IGN`) et l'exécution d'un traitant.
6. Après l'exécution d'un `setjmp`, quelle(s) contrainte(s) doivent respecter les intructions exécutées ultérieurement de façon à conserver la possibilité d'exécuter à tout moment un `longjmp`. Écrire un programme illustrant le dysfonctionnement lorsque la contrainte précédente n'est pas respectée.
7. Dans un contexte temps réel, un processus doit être capable de traiter des données dans un temps maximum. Lorsque le traitement est trop long un signal est envoyé au processus qui doit alors abandonner le traitement courant pour pouvoir acquérir de nouvelles données et les traiter. Implémenter ce mécanisme de traitement temps réel en utilisant les primitives systèmes introduites précédemment. On simulera simplement les aspects acquisition et traitement de données.

   Variante Réaliser un programme permettant de tester la réactivité d'un utilisateur. Le principe est le suivant : Le programme effectue une série de 10 tests. Un test consiste à
   — attendre pour une durée variable (en secondes, dans un premier temps).
   — afficher un message donnant un chiffre (variant pour chaque test) que l'utilisateur doit saisir.
   — à partir de ce moment, l'utilisateur dispose de deux secondes pour saisir ce chiffre au clavier.
      — si l'utilisateur a réagi dans les temps, cela est comptabilisé comme un succès et un message d'acquittement est affiché ;
      — sinon, un message d'échec est affiché.

   Lorsque l'ensemble des tests est fini, un message est affiché donnant le score global.

   Réaliser ensuite une seconde version, prenant en paramètre
   — le délai maximal de réaction (en millisecondes). Pour cela, utiliser des timers.
   — ainsi que le nombre de tests à effectuer.

   En outre, cette version
   — proposera un chiffre aléatoire dans le message d'invite,
   — et réalisera une attente aléatoire avant l'affichage du message d'invite.

   Pour cela, il est possible d'utiliser les fonctions `srandom` et `random` de `stdlib.h`
8. Ecrire la commande `timeout <temps> <commande> {<arg1>...<argn>}` qui respecte l'algorithme suivant : le processus père lance un processus qui exécute la commande dont le nom est passé en second paramètre avec éventuellement comme argument(s) le(s) paramètre(s) suivant(s) (`<arg1>...<argn>`)

puis arme une horloge de `<temps>` secondes. Le père protégé vis-à-vis de tous les signaux attend ensuite la fin du processus fils ou l'expiration de l'alarme. Si le processus fils se termine avant la fin du temps imparti, le père affiche le code retour du fils. Si le processus fils n'est pas terminé avant la fin du temps imparti, le père le tue et affiche (dans le `main`) un message indiquant que le délai de garde est échu.

Rappel : `atoi(argv[i])` renvoie la conversion de `argv[i]` en entier.

De même `ret=sscanf(argv[i],"%d",&val_num)` permet d'affecter la conversion de `argv[i]` à `val_num` avec les mêmes règles de retour que `scanf`.

9. Illustrer la différence entre le déroutement de signaux sur la fonction prédéfinie `SIG_IGN` et le masquage des signaux.
10. Un processus crée trois processus fils (F1,F2,F3). Les activités des quatre processus concurrents sont les suivantes :
    — le père attendra chacun des fils et effectuera un compte-rendu complet de leur exécution. L'attente sera réalisée par un `wait` puis un `pause` et enfin un `sleep` (avec un temps suffisant) ;
    — le premier fils F1 crée également un fils PF1 (*donc petit-fils du père*). F1 se met en attente à l'aide de `pause`. Le petit-fils PF1 après deux secondes tue son père F1 et se termine avec un code retour de 2 ;
    — le deuxième fils F2 après une attente de 10 secondes se termine avec un code retour de 3 ;
    — le troisième fils F3 après une attente de 15 secondes se termine avec un code retour de 6.

    L'ensemble des processus exécute une fonction `h_fils` lorsqu'un signal `SIGCLD` se produit. Cette fonction affiche le `pid` du processus qui a été interrompu puis le compte-rendu de l'exécution du processus qui a provoqué cette interruption.

2.6 Annexes

2.6.1 `strace` : traçage des appels systèmes

La commande `strace` (aussi dénommée `trace` ou `truss` selon les systèmes) appliquée à une commande imprime *tous* les appels systèmes effectués par cette commande.

3 Entrées-Sorties, synchronisation, communication

3.1 Les fichiers

Les fichiers sont identifiés au sein d'un processus par un entier appelé *descripteur* de fichier. Celui-ci est en réalité un simple indice dans un tableau de descripteurs gérés par le noyau. Ce tableau comporte un nombre limité d'entrées (fixé lors de l'installation du noyau, par exemple : 4096), ce qui restreint le nombre de fichiers ouverts simultanément par un processus. À la création d'un processus (`fork`), celui-ci hérite des descripteurs de fichiers de son père.
Par convention, les trois premiers descripteurs sont toujours ouverts au début d'un processus :

- 0 correspond à l'entrée standard (généralement le clavier) ;
- 1 correspond à la sortie standard (généralement l'écran) ;
- 2 correspond à la sortie standard des messages d'erreurs (généralement l'écran).

Dans `unistd.h`, trois macros définissent ces descripteurs : `STDIN_FILENO, STDOUT_FILENO` et `STDERR_FILENO`.

Les primitives `open, read, write` et `close` permettent respectivement la création, la lecture, l'écriture, et la fermeture de fichiers. Les primitives `dup` et `dup2` permettent la duplication de descripteurs. La primitive `pipe` permet d'établir une communication entre processus selon le schéma générique du producteur-consommateur. Les primitives `mkdir` et `rmdir` permettent respectivement la création et la destruction d'un répertoire. La primitive `unlink` permet de détruire un fichier.
Comme précédemment et sauf spécifié autrement, les primitives sont déclarées dans `unistd.h`.

3.1.1 Ouverture d'un fichier : `open`

```
#include <fcntl.h>
int open (char *nomf, int option, int mode);
```

La primitive `open` permet d'ouvrir l'accès à un fichier (et éventuellement de le créer s'il n'existe pas) de nom `nomf`. `nomf` constitue une référence relative par rapport au catalogue de travail du processus si cette chaîne ne commence pas par le caractère / et, sinon une référence absolue. La valeur des droits d'accès (`mode`) n'est prise en compte que si le fichier est effectivement créé (voir commande UNIX `chmod` pour les valeurs possibles de `mode`). `option` détermine le mode d'ouverture du fichier. Il peut prendre une ou plusieurs des constantes symboliques (séparées par des OU bit-à-bit, « | ») définies dans le fichier `fcntl.h`.

mnémonique	description
O_RDONLY	ouverture en lecture
O_WRONLY	ouverture en écriture
O_RDWR	ouverture en lecture et écriture
O_APPEND	ouverture en écriture en fin de fichier
O_CREAT	création du fichier avec droits d'accès définis par `mode`
O_EXCL	avec O_CREAT provoque un échec si le fichier existe déjà.
O_TRUNC	ramène la taille du fichier à zéro si le fichier existe déjà.

TABLE 3.1 – Valeurs pour le paramètre `option` de `open`

3.1.2 Masquage des protections : `umask`

`umask` permet de définir un masque de restriction appliqué lors de la création de fichiers (primitives `creat` ou `open` avec le mode `O_CREAT`). Les bits de `umask` spécifient les droits d'accès qui sont *refusés* pour les fichiers créés.
Exemple :

```
#include <sys/stat.h>
umask(0026)
open("FichTest", O_WRONLY|O_CREAT, 0666);
```

Le fichier `FichTest` est créé avec les droits d'accès : `[666 & ~(026)] = 640 = rw-r---`
Autrement dit, les droits d'accès sont, parmi ceux donnés par l'argument `mode`, ceux qui ne sont pas dans `umask` .

3.1.3 Fermeture d'un fichier : `close`

```
int close(int desc);
```

`close` ferme l'accès au fichier par le descripteur `desc` pour le processus appelant.

3.1.4 Duplication de descripteurs : `dup` et `dup2`

Il est possible de dupliquer un descripteur c'est-à-dire de créer un nouveau point d'accès à un fichier déjà ouvert.

```
int dup(int desc);
```

La primitive `dup` fournit un nouveau descripteur ayant exactement les mêmes caractéristiques que le descripteur `desc` (qui doit nécessairement exister). La primitive garantit que la valeur restituée est *la plus petite possible*. Autrement dit c'est le descripteur *libre* de plus petit indice qui est choisi et ré-alloué. La valeur de retour est -1 en cas de problème.

```
int dup2(int desc, int nv_desc);
```

La primitive `dup2` permet de forcer `nv_desc` comme synonyme de `desc`. Si `nv_desc` était un descripteur ouvert alors il est préalablement fermé. La valeur de retour est -1 en cas de problème.

Application : soit `desc` un descripteur de fichier ouvert en lecture, on souhaite que les prochaines lectures sur l'entrée standard se réalise dans le fichier associé à ce descripteur. Écrire à l'aide de `dup` (entre autres) les instructions répondant à cette spécification.

3.1.5 Accès aléatoire à un fichier : `lseek`

Les accès à un fichier se font selon le modèle d'une file séquentielle : un curseur est associé à chaque fichier, qui définit la position courante dans le fichier, à partir de laquelle se fera le prochain accès.
La primitive `lseek` permet de fixer arbitrairement la position d'accès courante.

```
long lseek(int desc, long offset, long origine);
```

`lseek` permet de déplacer le curseur associé au fichier accessible par `desc` de `offset` octets relativement à la position courante (`origine = SEEK_CUR`), au début de fichier (`origine = SEEK_SET`), ou à la fin du fichier (`origine = SEEK_END`). La valeur de retour de la fonction est la position du pointeur après déplacement (en nombre d'octets par rapport au début de fichier) ou -1 en cas d'erreur. Le début de fichier correspond à la position 0. Les valeurs possibles de `origine` (`SEEK_CUR`, `SEEK_SET` et `SEEK_END`) sont predéfinies dans le fichier `unistd.h`.

Remarques

1. la position courante du pointeur associé au fichier accessible par `desc` peut être obtenue par l'appel : `lseek(desc, 0, SEEK_CUR)` ;
2. si le fichier est accessible par plusieurs descripteurs (obtenus par la primitive open), la primitive `lseek` ne modifie, bien entendu, que le curseur correspondant au descripteur fourni en paramètre (ce n'est par contre pas le cas lors d'une duplication de descripteurs).

3.1.6 Lecture/écriture d'un fichier : `read/write`

```
int read(int desc, char *buf, int n_oct);
int write(int desc, char *buf, int n_oct);
```

```
#define Noctets 512                      |   #define Noctets 512
char buf[Noctets];                       |   char    *buf;
int desc, ret;                           |   int desc, ret;
                                         |   buf = (char *)malloc(Noctets);
ret = read(desc,buf,Noctets);            |   ret = read(desc,buf,Noctets);
/* equivalent a    */                    |
/*ret = read(desc,&(buf[0]),Noctets);*/ |
```

Cet appel permet la lecture de `Noctets` octets du fichier accessible via le descripteur `desc`, à partir de la position courante. La suite d'octets lus est placée dans le tampon `buf`. La position courante progresse du nombre d'octets lus.
La valeur de retour `ret` est le nombre d'octets effectivement lus ($\leq$ `Noctets`) ; elle vaut 0 si la fin de fichier est atteinte dès la lecture du premier octet ; elle vaut -1 en cas de problème (fichier fermé par exemple).

```
#define Noctets 512
char buf[Noctets];
int desc, ret;
ret = write(desc,buf,Noctets);
```

Cet appel permet d'écrire, à partir de la position courante, les `Noctets` premiers octets du tampon `buf` dans le fichier accessible via le descripteur `desc`. La valeur de retour est le nombre d'octets écrits ; elle vaut -1 en cas de problème. La position courante progresse du nombre d'octets écrits.
Ces deux fonctions peuvent être utilisées avec des enregistrements :

```
struct coord { int x; int y; } c1;
ret = read (desc, &c1, sizeof(struct coord));
```

3.1.7 Création/destruction d'un répertoire : `mkdir/rmdir`

```
#include <sys/stat.h>
int mkdir(char *nomr, int mode);
int rmdir(char *nomr);
```

La primitive `mkdir` permet la création d'un nouveau répertoire de nom `nomr` dont les droits d'accès sont, comme pour les fichiers, spécifiés par le paramètre `mode`. `rmdir` détruit le répertoire de nom `nomr`.

3.1.8 Désignation et destruction des fichiers

La désignation des fichiers unix utilise la notion de chemin d'accès via une suite de répertoires contenant des identifiants symboliques. La notion de lien permet de rendre accessible par plusieurs chemins un même fichier. La figure 3.1 illustre cette possibilité. Le fichier F est accessible par le chemin d'accès « étudiant » :

.../home/INFO/1AI/ETUD02/SO/exemple.c

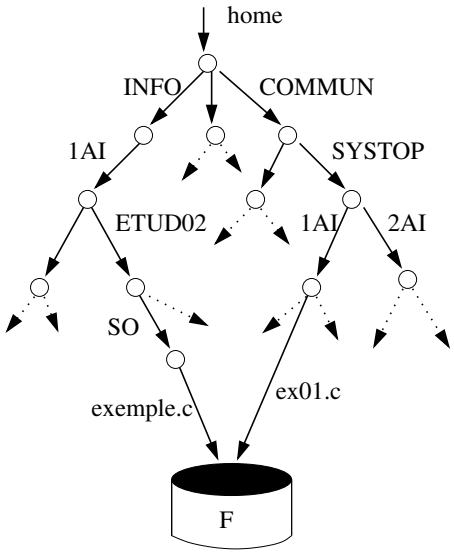


FIGURE 3.1 – Liens multiples sur un fichier

et par le chemin d'accès « professeur » :

.../home/COMMUN/SYSTOP/1AI/ex01.c

On constate que, par ce mécanisme de lien, l'arborescence de fichiers est en fait un graphe orienté sans cycle (DAG), les feuilles du graphe pouvant avoir plusieurs « pères ». La création d'un lien consiste à introduire une nouvelle entrée dans un répertoire, cette nouvelle entrée désignant un fichier existant. Par exemple, pour obtenir le graphe de la figure 3.1, si le fichier F existe sous le nom ex01.c, on peut créer le lien permettant de l'accéder sous le nom exemple.c en exécutant la commande suivante dans le répertoire courant .../home/INFO/1AI/ETUD02 :

ln .../home/COMMUN/SYSTOP/1AI/ex01.c SO/exemple.c

Une commande ln appelle une primive du noyau link. Nous présentons donc les primitives mettant en jeu les liens, autrement dit les entrées de répertoires.

Primitives link/unlink

```
int link(const char *fichier_existant, const char *nouveau_nom);
int unlink(const char *fichier_existant);
```

Sémantique Les paramètres fichier_existant et nouveau_nom spécifient deux noms de fichier éventuellement précédés par leur chemin d'accès. Dans les deux cas, le chemin d'accès doit exister au moment de l'appel. La primitive link crée un nouveau fichier dont le nom et le chemin d'accès sont spécifiés par nouveau_nom et le définit comme un lien vers le fichier dont le nom et le chemin d'accès sont spécifiés par fichier_existant. Inversement, la primitive unlink efface le nom du fichier existant spéficié de son répertoire.
La réussite de l'opération est tributaire des droits de parcours des chemins d'accès et du droit d'écriture dans le répertoire cible.
Ces primitives modifient un compteur de liens présent dans tout descripteur de fichier (i-node). La primitive link incrémente de 1 le compteur lorsqu'elle réussit et la primitive unlink le décrémente de 1. Lorsque ce compteur est à 0, cela signifie qu'il n'existe donc plus de répertoire contenant un nom permettant de désigner le fichier. Par conséquent, ce fichier sera automatiquement détruit par le noyau dès qu'il ne sera référencé par aucun répertoire.

Remarques

- la commande rm ne « détruit » pas obligatoirement un fichier. Elle se contente d'enlever un nom dans un répertoire (exécution d'une primitive unlink). C'est seulement lorsque ce nom est l'unique entrée de répertoire pointant vers le fichier que la commande entraîne tôt ou tard la destruction du fichier par le noyau de façon asynchrone ;
- le super-utilisateur a le droit de créer des liens sur des répertoires. Autrement dit, dans ce cas, la primitive link accepte en premier paramètre un répertoire. De même, la primitive unlink accepte un répertoire en opérande. Par exemple, cette possibilité est utilisée par la commande rmdir ;
- les liens ne peuvent exister que dans une arborescence de fichiers appartenant à un même système de fichiers (même volume support). Pour créer des liens entre deux systèmes de fichiers (autrement dit inter-volumes), il faut utiliser la notion de lien symbolique (voir commande ln avec l'option -s).

Exemple : fichier temporaire disparaissant à la terminaison du processus (quelle que soit la cause de terminaison) :

```
int desc_tmp;
desc_tmp = open ("/tmp/toto", O_WRONLY | O_CREAT | O_TRUNC, 0777);
unlink("/tmp/toto");
.......
close(desc_tmp);
```

3.1.9 Différences entre `open`, `dup` et `dup2`

Les différences entre les primitives `open`, `dup` et `dup2` du point de vue de la manipulation des tables du système sont résumées dans la figure 3.2.
L'ouverture d'un fichier se traduit par l'allocation d'un descripteur dans un tableau de descripteurs de « fichiers ouverts » associé au processus appelant [1]. L'indice du descripteur alloué est renvoyé comme moyen d'accès au fichier désormais ouvert. Ce descripteur pointe lui même un descripteur de fichier « utilisé » qui contient notamment le curseur courant de lecture/écriture associé au fichier utilisé. Ce descripteur de fichier utilisé peut être partagé par plusieurs processus, par exemple entre un processus père et un processus fils. Enfin, ce descripteur pointe lui-même une copie du i-node associé au fichier. L'ouverture assure la création de ce chemin d'accès au fichier cible.
L'opération dup2 ne fait que dupliquer la liaison par l'allocation d'un nouveau descripteur de fichier ouvert en tout point semblable au descripteur initial. On possède désormais deux canaux d'accès pour lire ou écrire dans le même fichier. Il faut bien noter que le curseur de lecture ou d'écriture est partagé.
Une deuxième opération d'ouverture du même fichier engendre par contre non seulement l'allocation d'un descripteur de fichier ouvert (premier niveau) mais aussi la création d'un descripteur de fichier utilisé. Par conséquent, une lecture du fichier F1 via le canal de numéro d1 ne lira pas forcément la même zone de fichier qu'une lecture via le numéro d3. Ici, le curseur n'est plus partagé.
Enfin, lors d'une opération fork, le processus fils hérite de tous les canaux ouverts par le processus père. En particulier, le fils est créé avec une copie de la table des fichiers ouverts du processus père, et partage donc les fichiers utilisés avec son père.

1. Bien que l'on conserve la terminologie classique de « fichier ouvert », il vaudrait mieux parler de flot de données, la cible ou la source des données pouvant être une connexion réseau par exemple.

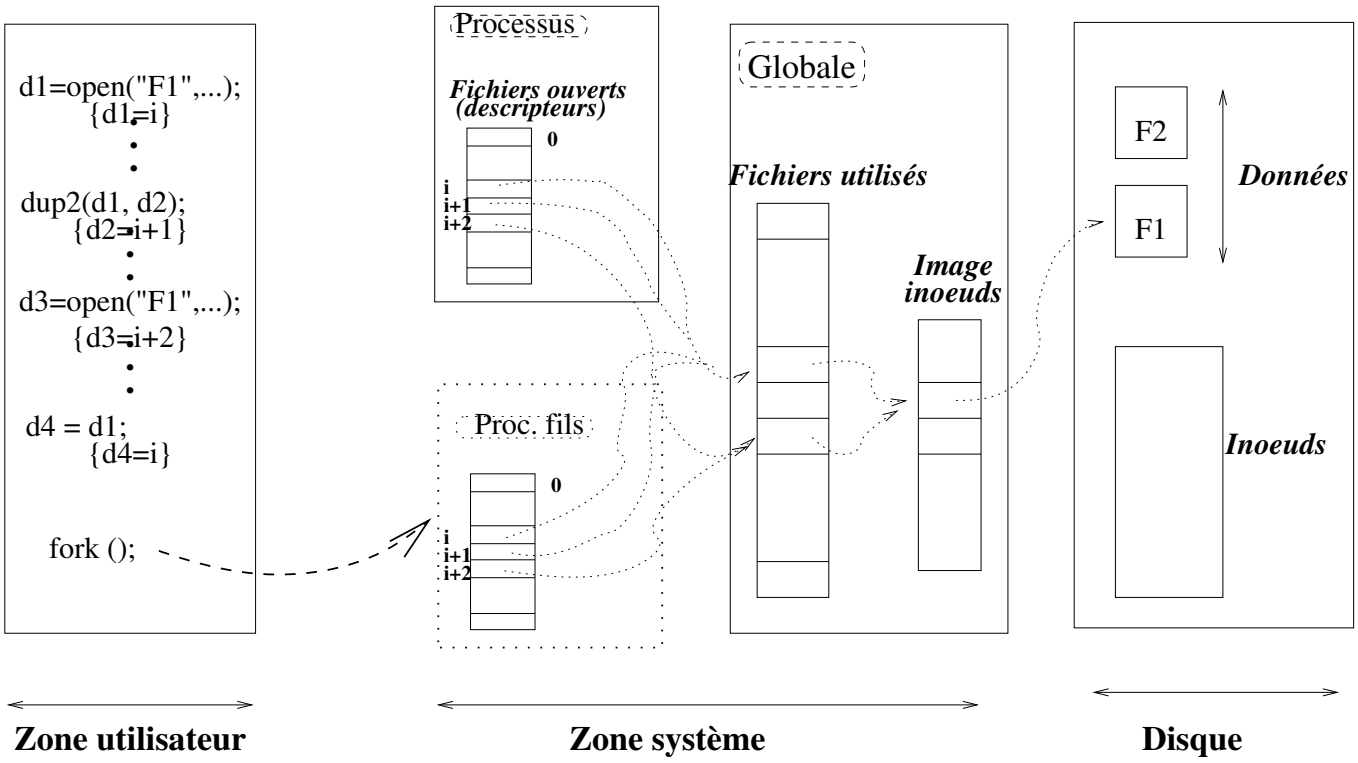


FIGURE 3.2 – Manipulation des descripteurs et tables du système

3.2 Communication par tubes

Les tubes (*pipes*) permettent de faire communiquer des processus selon le schéma générique du producteur-consommateur. Un tube se comporte comme une file (FIFO) de *capacité limitée* où les informations sont introduites à une extrémité et sont extraites à une autre. Un tube est implémenté comme un fichier sans nom et non permanent.
La création d'un tube correspond à celle de deux descripteurs de fichiers, l'un permettant d'écrire dans le tube, l'autre d'y lire par les opérations `read` et `write` précédemment décrites.
La capacité d'un tube est de 8K octets. Si l'on continue à écrire dans un tube plein, le processus rédacteur est bloqué. Si un processus lit dans un tube vide et si au moins un processus peut écrire dans ce tube, le lecteur est bloqué tant que le tube reste vide.

3.2.1 La primitive `pipe`

La primitive `pipe` permet la création d'un tube de communication.

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

Au retour de l'appel :
- `desc[0]` est la sortie du tube, i.e. le numéro du descripteur pour la lecture (mnémonique : 0 est l'entrée standard) ;
- `desc[1]` est l'entrée du tube, i.e. le numéro du descripteur pour l'écriture (mnémonique : 1 est la sortie standard) ;
- la valeur de retour est 0 si la création s'est bien passée et -1 en cas de problème.

Comme pour les autres descripteurs, il y a héritage des tubes créés lors de l'exécution de la primitive `fork` (et conservation lors de `exec`). Deux processus pourront donc communiquer via un tube si ce tube a été créé par un ancêtre commun. Le système assure de plus certains contrôles :

— si un processus tente d'écrire dans un tube alors qu'aucun processus n'est en mesure d'y lire, le signal `SIGPIPE` est envoyé au processus. Le processus est alors interrompu s'il ne traite pas le signal en utilisant la primitive `signal` ;
— la primitive `read` sur un tube a pour valeur 0 (« fin de fichier ») si le tube est vide et que plus aucun processus n'est en mesure d'écrire dans le tube.

3.2.2 Exercice : producteur-consommateur

Définir deux processus devant respectivement :

1. *Un producteur, processus 1* : écrire dans un pipe les nombres 1 à N ;
2. *Un consommateur, processus 2* : lire les nombres écrits dans ce pipe, en faire la somme puis afficher sur la sortie standard le résultat. Ce processus ne connait pas la valeur de N.

Une fois cette version implantée, on souhaite la compléter de deux façons (indépendantes) :

1. ajouter un processus qui effectuera un compte-rendu de l'exécution des deux processus producteur et consommateur : causes des terminaisons, et codes retour ;
2. le processus 2 écrit maintenant son résultat dans le pipe et on ajoute un processus qui lit ce résultat et l'affiche sur la sortie standard. Quel(s) problème(s) pose(nt) cette spécification ?

3.3 Contrôle du mode d'exécution des échanges de flots de données

3.3.1 Échanges en mode bloquant ou immédiat

Lorsqu'on ouvre un flot de données (en entrée ou en sortie) sur une ressource (fichier, pipe, socket,...), les primitives d'échange read et write s'exécutent par défaut en mode bloquant. Cela signifie qu'une opération de lecture ou d'écriture se termine seulement lorsque l'opération, si elle n'était pas erronée, a pu être correctement exécutée. Autrement dit, côté appelant, la terminaison sans erreur d'une telle opération permet de garantir que l'opération a été exécutée correctement et complètement. Cependant un tel mode de fonctionnement implique un blocage possible du processus appelant jusqu'à ce que l'opération puisse être exécutée. À titre d'exemple, s'il s'agit d'une lecture d'une ligne frappée au clavier, il faut attendre qu'une telle ligne ait été frappée et validée par l'usager. S'il s'agit d'une écriture sur un pipe, il faut éventuellement attendre que le pipe ne soit plus plein.
Il est possible d'envisager un deuxième mode d'exécution évitant tout blocage. Il s'agit donc d'un mode immédiat : le noyau essaie d'exécuter immédiatement l'opération demandée. Cependant, elle n'est pas toujours possible et par conséquent, une opération non erronée peut se terminer de trois façons :

1. sans avoir réalisé l'échange demandé : en lecture, les données n'étaient pas disponibles ; en écriture, il n'existait pas d'espace tampon libre ;
2. en ayant partiellement réalisé l'opération : en lecture, des données étaient disponibles mais en taille insuffisante ; en écriture, l'espace tampon libre était insuffisant ;
3. en ayant complètement réalisé l'échange : cas identique au mode bloquant.

Bien sûr ce mode d'exécution immédiat (non bloquant) est plus difficile à utiliser pour le programmeur. Il doit prévoir les conditions nouvelles de terminaison. L'avantage de ce mode reste néanmoins le non blocage du processus appelant qui peut continuer son exécution même si l'opération d'échange ne s'est pas complètement exécutée : il sera possible de tenter à nouveau un appel ultérieurement. La primitive fcntl permet de commuter les échanges sur un flot du mode bloquant au mode immédiat et inversement.

3.3.2 Adaptation à l'environnement de communication

Lorsqu'un processus communique via plusieurs flots de données entrants ou sortants, la possibilité d'exécuter un échange est conditionnée par le comportement des processus partenaires. Par exemple, supposons qu'un

processus ait ouvert deux flots de données entrants et un flot sortant via des pipes. La figure 3.3 illustre un tel environnement de communication. Le processus P0 peut selon l'état des pipes exécuter des opérations de lecture ou écriture de données. S'il exécute un algorithme de scrutation des pipes dans l'ordre $i, i+1, i+2$, il risque alors de rester bloqué sur une opération du flot i parce que le pipe correspondant est vide ($P1$ ne produit pas beaucoup de données) alors qu'il pourrait soit écrire via le flot $i+1$ ou lire via le flot $i+2$ parce que le pipe correspondant est, pour le flot de sortie, non plein et, pour le flot d'entrée, non vide.
On constate donc qu'il serait plus intéressant d'avoir la possibilité de tester et, éventuellement d'attendre, qu'un échange au moins soit possible via l'un des flots. Il faut pour cela une primitive spécifique permettant de tester l'état d'un ensemble de flots de données. Le noyau Unix dispose d'une telle primitive appelée select. Dans l'exemple précédent, son appel permet de tester si des échanges sont possibles sur chacun des trois flots. Après une période d'attente éventuelle, la primitive se termine en indiquant le(s) flot(s) sur le(s)quel(s) une opération est possible. On peut alors à coup sûr et sans entraîner de blocage du processus $P0$, exécuter un échange sur le(s) flot(s) selectionné(s) par l'opération select.

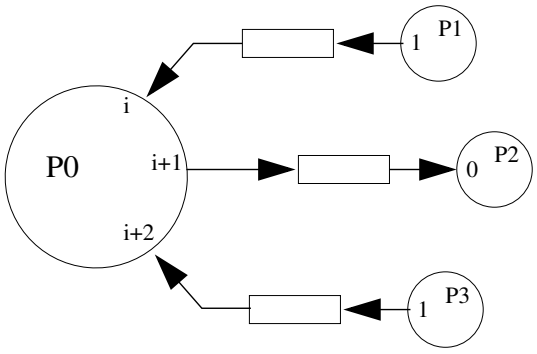


FIGURE 3.3 – Un environnement de communication par pipes

3.3.3 La primitive `fcntl`

La primitive `fcntl` permet de contrôler et modifier l'état d'un descripteur de fichier (connecté à un fichier, un tube ou un socket) :

```
#include <fcntl.h>
#include <sys/types.h>
int fcntl(int desc, int cmd, int arg);
```

`cmd` décrit la commande effectuée par `fcntl` sur le descripteur ouvert `desc`. `arg` est utilisé comme paramètre de certaines des commandes à effectuer.
Valeurs possibles de `cmd` :

- `F_DUPFD` : la valeur de retour est alors un nouveau descripteur avec les mêmes caractéristiques que `desc`. Cette valeur est égale à la plus petite des valeurs de descripteurs disponibles plus grandes que `arg`.
- `F_GETFL` : la valeur de retour indique en retour l'état du descripteur `desc`. Les valeurs possibles sont :
  ```
  O_RDONLY
  O_WRONLY
  O_RDWR
  O_NONBLOCK /* entree/sortie non bloquante, = O_NDELAY */
  ```
- `F_SETFL` : `arg` est positionné pour le descripteur `desc`. Si `arg = O_NDELAY` ou `O_NONBLOCK`, alors l'accés à `desc` est non bloquant.

3.3.4 Exemple : Lecture non bloquante de données au clavier

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

#define LMAX 64

int desc_non_bloquant(int desc)
{
    /* -icanon : pour quitter le mode canonique, ie ne pas attendre
     *           le retour-chariot
     * min     : nombre de caracteres a recevoir au minimum avant de
     *           les transmettre au processus les lisant */
    system("stty -icanon min 1");
    return(fcntl(desc, F_SETFL, fcntl(desc, F_GETFL) | O_NONBLOCK));
}

int main(void)
{
    char buf[LMAX];
    int  flag, nlu;

    flag= fcntl(0, F_GETFL, 0);
    printf("flag initial : %d\n", flag);

    if (desc_non_bloquant(0) == -1) {
        fprintf(stderr,"Impossible de passer en non bloquant\n");
        exit(1);
    }

    flag= fcntl(0, F_GETFL, 0);
    printf("flag apres modification : %d\n", flag);

    printf("Une p'tite attente\n");
    sleep(3);
    printf("C'est parti\n");

    while(1) {
        nlu= read(0,buf,LMAX);

        if ( nlu <= 0 ) {
            write(1, "attente...\n", 11);
            sleep(1);
        } else {
            write(1, buf, nlu);
        }
    }
    return(0);
}
```

3.3.5 La primitive `select`

La primitive select permet de vérifier si les ressources nécessaires à l'exécution d'un opération d'échange de données sont disponibles (données en lecture, espace tampon en écriture) sur un ensemble de descripteurs précisés en paramètres (le primitive poll peut également être utilisée). Cette primitive permet aussi de tester si une condition exceptionnelle est détectée sur un flot (Exemple : message urgent en attente sur une communication par socket).
La primitive select possède trois modes d'exécutions possibles selon la valeur du dernier paramètre timeout.

1. test immédiat de l'état courant des descripteurs si timeout a pour valeur 0 ;
2. attente limitée éventuelle si timeout a une valeur différente de 0 ;
3. attente non bornée si timeout est omis (NULL).

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
int select (int nbdesc, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
            struct timeval *timeout)
```

Le type `fd_set` décrit un tableau de 1024 bits. Le bit numéro `idesc` d'une variable de type `fd_set` est associé au descripteur de fichier `idesc`. `select` regarde les descripteurs de numéro 0 à `nbdesc-1` correspondant à des bits positionnés à 1 dans les tableaux pointés par `readfds, writefds`, et `exceptfds` et vérifie si de l'information est disponible respectivement en lecture, en écriture ou si une exception a été levée. La valeur de retour indique le nombre total de descripteurs prêts (le bit correspondant reste à 1). Si cette valeur est nulle alors le temps indiqué dans `timeout` a été atteint et aucun descripteur prêt n'a été détecté. Si la valeur de retour est égale à -1 alors une erreur a été détectée par `select`. Dans ce cas le contenu des descripteurs en sortie n'est pas interprétable. La variable `errno` décrit alors la nature de l'erreur et contient la dernière erreur système s'étant produite (le message associé peut être imprimé en utilisant `perror`, voir exemple suivant).

- Valeurs possibles de `errno` en retour de `select`
  - `EBADF` : un des descripteurs positionnés dans un des ensembles de descripteurs est invalide ;
  - `EFAULT` : erreur dans le passage des arguments du `select` ;
  - `EINTR` : select interrompu par l'arrivée d'un signal ;
  - `EINVAL` : valeur de timeout non correcte.
- Macros permettant de manipuler des données de type `fd_set` (ensemble des 1024 descripteurs de fichiers).

```
int fd;
fd_set fdset;
FD_ZERO (&fdset)      /* initialise un ensemble de descripteurs fdset a 0 */
FD_SET (fd, &fdset)   /* positionne le descripteur fd de l'ensemble fdset */
FD_CLR (fd, &fdset)   /* supprime le descripteur fd de fdset */
FD_ISSET (fd, &fdset) /* valeur de retour non nulle si fd est dans fdset */
                      /* valeur de retour nulle sinon */
```

3.3.6 Exemple d'illustration des fonctionnalités du select.

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

#define NMAX 10000
```

```
#define NBDESC FD_SETSIZE

int main(void)
{
    int res, nlu;
    int f1, f2;
    char buf[NMAX];
    fd_set readfds, writefds;
    struct timeval timeout;

    timerclear(&timeout);
    timeout.tv_sec=3;
    timeout.tv_usec=10;
    if ((f1=open("FICH_TEST", O_CREAT |O_RDWR ,0666))==-1) {
        perror ("Erreur ouverture FICH_TEST");
        exit(1);
    }
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);

/* test 1 */
    printf("1/ ... select: Lecture sur stdin et sur fichier ouvert?\n");
    FD_SET(0, &readfds);  /* on ecoute sur l'entree standard */
    FD_SET(f1, &readfds); /* on ecoute sur le fichier f1 */
    res = select (NBDESC, &readfds, &writefds, NULL, &timeout);
    printf("  valeur de retour de select %d\n",res);
    if (res==-1) {
        perror("erreur retour de select");
    } else {
        if (FD_ISSET(f1,&readfds)==0) printf("   rien de recu sur desc f1\n");
        if (FD_ISSET(0,&readfds)==0) printf("   rien de recu sur desc 0\n");
    }
/* test 2 */
    printf("2/ ... select: Lecture sur stdin et Ecriture sur fichier ouvert?\n");
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
    FD_SET(0,&readfds);
    FD_SET(f1,&writefds);
    sleep(4); /* attente */
    res = select (NBDESC, &readfds, &writefds, NULL, &timeout);
    printf("  valeur de retour de select %d\n",res);
    if (res==-1) {
        perror("erreur retour de select");
    } else {
        if (FD_ISSET(f1,&writefds)==0) printf("   ecriture impossible sur desc f1\n");
        if (FD_ISSET(0,&readfds) != 0) {
            nlu = read(0,buf,NMAX);
            printf("  message: %20s recu sur desc 0\n",buf);
        }
    }
/* test 3 */
    printf("3/ ... select: sur descripteur ne correspondant pas a un fichier?\n");
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
    FD_SET(0,&readfds);
    FD_SET(f1,&readfds);
    f2 = 244; /* test de descripteur ne correspondant pas a un fichier ouvert */
```

```
    FD_SET(f2,&readfds);
    res = select (NBDESC, &readfds, &writefds, NULL, &timeout);
    printf("  valeur de retour de select %d\n",res);
    if (res==-1) {
        perror("   erreur detectee au retour du select"); /* (errno==EBADF)*/
    } else {
        if (FD_ISSET(0,&readfds)==0) printf("   rien de recu sur desc 0\n");
        if (FD_ISSET(f1,&readfds)==0) printf("   rien de recu sur desc f1\n");
    }
}
```

Résultat obtenu :

```
 1/ ... select: Lecture sur stdin et sur fichier ouvert?
   valeur de retour de select 1
   rien de recu sur desc 0
 2/ ... select: Lecture sur stdin et Ecriture sur fichier ouvert?
 donnee saisie au clavier
  valeur de retour de select 2
  message: donnee saisie au clavier
  recu sur desc 0
 3/ ... select: sur descripteur ne correspondant pas a un fichier?
   valeur de retour de select -1
   erreur detectee au retour du select: Bad file number
```

3.4 Exercices

1. Écrire un programme `ex_cat` qui permet de réaliser la commande `cat` UNIX. Comme pour la commande `cat`, une liste de fichiers et l'entrée par défaut doivent pouvoir être acceptées.
2. Écrire un programme qui réalise la commande UNIX `cp` (`cp source destination`) en utilisant le programme `ex_cat`. On souhaite de plus que le ficher destinataire de la copie soit uniquement accessible en lecture pour tous les utilisateurs (groupe et autres).
3. Écrire un programme qui utilise le mécanisme de pipes pour réaliser la commande :
   `who | grep nom_utilisateur | wc -l`
4. Écrire/valider la fonction :
   `int copy_bytes (int from, int to, int nbbytes)`
   qui copie `nbbytes` octets depuis un descripteur `from` sur le descripteur `to`. Le paramètre de retour est 0 si la copie s'est bien passée, et vaut -1 en cas déchec.
5. Écrire le code suivant : deux processus (père et fils) écrivent dans un même fichier, effectuent d'autres opérations (par exemple un sleep), puis relisent l'information écrite. Quels problèmes peuvent se produire à l'exécution ? Quelle est l'influence de la manière dont les deux processus ont obtenu un descripteur sur le fichier (open puis fork ou fork puis open dans chaque processus) ? Illustrer en travaux pratiques.
6. Écrire le code suivant : un processus (père) crée un répertoire de travail (RESTMP) dans lequel ses fils vont créer des fichiers de travail et y écrire. Une fois les processus fils terminés, le processus père doit nettoyer le répertoire RESTMP, puis le détruire.
7. Crible d'Eratosthène parallèle. Le but de cet algorithme est de déterminer les nombres premiers infèrieurs à un entier donné n. On définira dynamiquement une chaîne de processus. Le premier processus émet les nombres entiers de 3 à $n-1$. Chaque fois qu'un nouveau nombre premier m est détecté un nouveau processus est ajouté en bout de chaîne pour filtrer les multiples de m. La communication entre les processus sera réalisée au moyen de tubes.
8. Écrire les trois versions de l'énoncé ci-dessous.

   Version 1 : on dispose d'un exécutable `transmet_v1` qui considère en argument un nombre entier et à l'aide d'un `write` écrit ce nombre sur la sortie standard. Si le nombre d'arguments ou la valeur de l'argument n'est pas correct, il écrit `-1`.

   Un processus crée trois processus et associe à chacun d'eux un tube pour la communication. Chaque fils exécute `transmet_v1` avec comme paramètres : 1 et 2 pour le premier fils, 10 pour le 2ème, 7 pour le dernier. Le père s'endort deux secondes puis se met en écoute (au plus 3 secondes) sur les descripteurs dédiés à la communication avec ses fils. Il récupère ensuite les données éventuellement disponibles.

   Vous implanterez `transmet_v1.c` puis les codes du père et de ses fils.

   Version 2 : on modifie `transmet_v1.c` : les fils s'endorment, si le paramètre est correct (valeur entière), de la valeur du paramètre avant de l'écrire sur la sortie standard (`transmet_v2.c`).

   Le père scrute au plus trois secondes les descripteurs, et traite les données éventuellement disponibles. Il recommence ce traitement jusqu'à ce que toutes les écritures des fils aient été traitées.

   Version 3 : dans la version 2, on ne traite pas la terminaison des fils. En acceptant l'hypothèse que chaque fils exécute une écriture puis réalisera une tache particulière, comment faire pour que le père détecte la terminaison des fils (cause, code retour...) au moment adéquat ? Codez cette dernière version en prenant en compte la réponse à la question précédente (`transmet_v3.c`).
9. Écrire le code suivant : un processus père lit les données saisies sur l'entrée standard (en mode non bloquant) de facon répétitive sur une période de 20 secondes. Le père envoie au fur et à mesure de leur disponibilité les données à un processus consommateur fils qui affichera la liste des messages recus en fin de traitement.

10. Un processus père consomme et affiche de l'information de provenance multiple :
    (a) l'entrée standard ;
    (b) un fils envoie des données au travers d'un pipe avec un débit assez lent ;
    (c) un fils plus bavard envoie des données sur un autre pipe.

    Utiliser le select pour rendre le processus père aussi efficace que possible dans la consommation d'informations. Le père doit lister les messages recus en précisant leur provenance. La fin du père est déterminée par la fin de ses deux fils.
11. Compléter l'exercice 7 de la section 2.5 en supposant maintenant que le processus temps réel consomme des données produites par un autre processus. Un tube (*pipe*) sera utilisé pour établir la communication. Écrire le code du producteur (il doit produire à intervalles réguliers les données) et adapter le code du processus de traitement des données.
12. On considère le programme suivant :

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc,char *argv[])
{
  int p[2], pid;
  pipe(p);
  if ( (pid=fork()) == 0 ) {
     int n, doub_n;
     pid = fork();
     read(p[0], &n, sizeof(int));
     close(p[0]);
     doub_n = 2*n;
     printf("processus %d n=%d doub_n=%d\n", pid, n, doub_n);
     write(p[1], &doub_n, sizeof(int));
     close(p[1]);
  } else {
     int tab[5], nb1, nb2, i, j;
     nb1=10;
     nb2=40;
     printf("nb1 = %d nb2 = %d\n", nb1, nb2);
     write(p[1], &nb1, sizeof(int));
     write(p[1], &nb2, sizeof(int));
     close(p[1]);
     i = 0;
     while ( read(p[0], &(tab[i]), sizeof(int)) > 0 )
         i++;
     close(p[0]);
     for (j=0; j<i; j++)
         printf("%d\n", tab[j]);
  }
  return 0;
}
```

    (a) décrire précisément les fonctionnalités du programme précédent ;
    (b) lors d'une exécution de ce programme, le processus fils lit sur le pipe la valeur « 80 ». Expliquer ;
    (c) que se passe-t-il si l'instruction `close(p[1])` dans le processus père est supprimée ?
    (d) une exécution de ce programme peut entraîner le blocage du père et d'un de ses fils. Expliquer.
13. Exécuter le programme suivant en l'appelant avec un paramètre. Si l'exécutable s'appelle `ff`, faire par exemple `ff a`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(int argc,char *argv[])
{
 int i,d, nf, np, etat;

 d = 1;
 for (i=1; i<=argc; i++)
     {
        printf("chez %i : execution d'un fork()\n", (int) getpid());
        np = fork();
        if (np == 0)
           d = i+1;
     }
 printf("chez %i la variable d vaut %i\n", (int) getpid(), d);
 for (i=d; i<=argc; i++)
     { nf = wait(&etat);
       printf("chez %i : fin de %i\n",(int) getpid(),nf);
     }
 return 0;
 }
```

Compter le nombre de lignes imprimées en sortie à l'écran, comparer le avec le résultat de la commande `ff a | wc -l` et expliquer la différence.