ECOLE NATIONALE SUPERIEURE D'INFORMATIQUE ET DE MATHEMATIQUES APPLIQUEES DE GRENOBLE

Institut National Polytechnique de Grenoble

VOUS AVEZ DIT:
IPC SOUS UNIX SYSTEM V?

Olivier BERNARD & Bernard BOLZ & Thierry ELSENSOHN
Année Spéciale Informatique 1990-1991

Document réalisé sous la responsabilité de Serge ROUVEYROL.

Laboratoire LGI-IMAG
BP 53X
38041 GRENOBLE CEDEX

28 Juin 1991

REMERCIEMENTS

Pour les nombreux cafés et les cigarettes qu'il a partagés avec les deux tiers de l'équipe, et pour les idées qu'il nous a apportées, nous remercions Serge ROUVEYROL, à qui nous souhaitons l'éternité.
Nos remerciements vont également à Bernard CASSAGNE, dont les conseils ont toujours été extrêmement précieux.

Contents

Introduction

L'objet de ce document est de fournir des éclaircissements concernant l'utilisation de certaines primitives et de certains objets de programmation UNIX de la version System V. Les différents programmes fournis dans chaque chapitre devraient permettre une bonne compréhension des sujets abordés.

Dans un premier temps, nous procédons à une description non exhaustive des primitives permettant de gérer les entrées-sorties et les processus. Cette description apporte les connaissances suffisantes qui permettent d'appréhender les sujets traités par la suite.

La deuxième partie est consacrée à l'étude des mécanismes de base de communication entre processus, que sont les signaux et les conduits.

La troisième et dernière partie met en évidence les particularités des extensions propres à System V, et communément appelées IPC (Inter-Process Communication), que constituent les sémaphores, la mémoire partagée et les messages. Ces trois concepts sont autant de mécanismes avancés de communication inter-processus.

Ce document, ainsi que les exemples, sont disponibles via `ftp` anonyme sur la machine `imag.imag.fr`[1] (lire le fichier `/pub/DOC.UNIX./IPC.SV/README`).
Toute remarque concernant ce document peut être adressée à `serge@imag.fr`.

[1]Toute reproduction, duplication, modification de ce manuel à des fins personnelles ou professionnelles est fortement conseillée, en particulier pour les étudiants pauvres.

Part I

NOTIONS DE BASE

Chapter 1

GENERALITES

1.1 Appels système – Routines de librairies

Tout d'abord, il est nécessaire de faire la distinction entre un appel système (primitive) et une routine de librairie.
Lors d'un appel système, l'utilisateur demande au système de réaliser une opération en son nom. Par exemple, `read()` est un appel système qui demande au système de remplir un tampon avec des données enregistrées sur périphérique.
Une routine de librairie, en revanche, ne requiert habituellement pas le système pour réaliser l'opération désirée. C'est le cas de la fonction `sin()` qui est calculée par sommation d'une série finie, alors que la fonction `popen()`, elle, est un sous-programme de la librairie qui permet d'ouvrir un conduit en spécifiant la commande à exécuter, et utilise pour cela les primitives `pipe()`, `fork()`, `open()` et `close()`. Les primitives sont expliquées dans le manuel UNIX numéro 2, alors que les routines de librairie le sont dans le manuel numéro 3.

1.2 Gestion des erreurs

Toutes les fonctions dont nous allons parler rendent une valeur en retour. La plupart du temps, en cas d'erreur lors de l'exécution d'une primitive, la valeur retournée vaut -1. Dans ce cas, la variable globale `errno` est mise à jour, et contient le code d'erreur précis. On peut visualiser ce code à l'aide de la fonction `perror()` qui donne son libellé en clair. Il est nécessaire d'inclure le fichier `<errno.h>` au programme pour pouvoir utiliser `errno` et `perror()`.
On trouvera en annexe les codes les plus couramment rencontrés.

1.3 Fonctionnement de man

Il existe sur le système un manuel définissant toutes les primitives. On peut y accéder par la commande shell:

```
man nom_de_primitive
```

L'aide correspondante apparaît alors à l'écran.

On peut également imprimer cette aide de la façon suivante:

1. Il faut tout d'abord se placer dans un des répertoires `/usr/catman/X_man/`, où `X_man` correspond à l'un des quatre répertoires suivants:
   - `p_man` : programmeur.
   - `u_man` : utilisateur.
   - `a_man` : administration.
   - `i_man` : réseau.
2. Les fichiers se présentent alors sous deux formes: compactée (`nom.2.z` par exemple), ou non compactée (`nom.2` par exemple).
   Dans le premier cas, il faut d'abord décompacter le fichier vers un fichier temporaire, grâce à la commande:
   ```
   pcat nom.2.z > /tmp/nom
   ```
   Dans le second cas, il faut rediriger le fichier vers un fichier temporaire :
   ```
   cat nom.2 > /tmp/nom
   ```
3. Il faut ensuite imprimer le fichier temporaire à l'aide des commandes d'impression classiques.

Notons que ce qui vient d'être décrit est valable sur les DPX de l'E.N.S.I.M.A.G.

1.4 Informations concernant les utilisateurs

On trouvera les informations concernant les utilisateurs dans les fichiers `/etc/passwd`, `/etc/utmp` et `/etc/group`.

1.4.1 Login

Chaque utilisateur du système dispose d'un nom de `login` unique qui permet de l'identifier. Ce nom est constitué de 8 caractères au plus. Il est seulement utilisé par les programmes de niveau utilisateur (la messagerie électronique entre autres). Le noyau du système, lui, ne l'utilise pas.

L'appel système `getlogin()` permet d'obtenir le `login` de l'utilisateur exécutant un programme en recherchant, dans le fichier `/etc/utmp`, le terminal sur lequel il s'exécute, et en retournant le `login` associé à ce terminal.

Valeur retournée: pointeur vers une chaîne de caractères ou NULL en cas d'échec.

1.4.2 Description du système de droits d'accès sous UNIX

Chaque fichier possède, dans son nœud d'index (voir en 1.5.1), un identificateur d'utilisateur et un identificateur du groupe du propriétaire (voir en 1.4.3). Le nœud d'index contient également un nombre binaire de 16 bits, dont les neuf premiers (les plus à droite) constituent les droits d'accès interprétés comme autorisation en lecture, écriture, et exécution pour le propriétaire, le groupe, ou les autres usagers. Un bit est positionné à 1 si le droit d'accès est accordé, et à 0 sinon.
Ces neuf premiers bits sont détaillés ci-dessous:

Droits d'accès	
numéro du bit	fonction
8	lecture par le propriétaire
7	écriture par le propriétaire
6	exécution par le propriétaire
5	lecture par les membres du groupe
4	écriture par les membres du groupe
3	exécution par les membres du groupe
2	lecture par les autres usagers
1	écriture par les autres usagers
0	exécution par les autres usagers

Pour modifier ces droits d'accès, on utilise la commande shell `chmod`. On pourra préciser les options suivantes:

- `o` : pour modifier uniquement les 3 bits *autres usagers*
- `g` : pour modifier uniquement les 3 bits *membres du groupe*
- `u` : pour modifier uniquement les 3 bits *utilisateur* (propriétaire du `login`).

Si rien n'est précisé, tout le monde est affecté par la modification.

Exemple:

```
obernard/> ll
total 3
-rw-rw-rw-   1 obernard speinf        16 Jun 11 15:45 fichier1
-rw-rw-r--   1 obernard speinf        18 Jun 11 15:46 fichier2
-rw-rw-r--   1 obernard speinf        13 Jun 11 15:46 fichier3
-rw-rw-r--   1 obernard speinf        12 Jun 11 15:47 fichier4
obernard/>chmod -w fichier1
obernard/>chmod o-r fichier2
obernard/>chmod g+x fichier3
```

```
obernard/>chmod u+x fichier4
obernard/>ll
total 3
-r--r--r--   1 obernard speinf        16 Jun 11 15:45 fichier1
-rw-rw----   1 obernard speinf        18 Jun 11 15:46 fichier2
-rw-rwxr--   1 obernard speinf        13 Jun 11 15:46 fichier3
-rwxrw-r--   1 obernard speinf        12 Jun 11 15:47 fichier4
```

Lors de l'utilisation de primitives d'entrées/sorties, il est parfois nécessaire d'indiquer les droits d'accès associés au fichier, en utilisant un entier obtenu en convertissant la valeur binaire en octal.

Par exemple, pour avoir l'autorisation en lecture, écriture et exécution pour le propriétaire et le groupe, et l'autorisation en lecture et en exécution pour les autres, il faudra utiliser lors de l'appel de la routine de création du fichier:

111111101 sous forme binaire

775 sous forme octale

Dans les sept bits restants, deux concernent la modification d'identité de l'utilisateur (bit 11 set-uid), et la modification de l'identité du groupe (bit 10 set-gid). Lorsque le bit 11 (respectivement le bit 10) est positionné à 1, l'indicateur de permission d'exécution pour le propriétaire (respectivement le groupe) visualisé par la commande `ls` est à `s` (voir l'exemple ci-dessous).

Prenons l'exemple d'un programme ayant le bit `s` (s pour set-uid-bit) positionné pour le propriétaire. A l'exécution d'un tel programme, l'utilisateur effectif est changé en le propriétaire du fichier contenant le programme. Puisque c'est l'utilisateur effectif qui détermine les droits d'accès (voir en 1.4.3), cela permet à un utilisateur d'acquérir temporairement les droits d'accès de quelqu'un d'autre.
Cette remarque s'applique aussi pour les identificateurs de groupe.
C'est ce mécanisme qui permet à tout utilisateur de modifier le fichier `/etc/passwd` dont l'utilisateur privilégié, `root`, est propriétaire, au travers de la commande `passwd()`. Le bit `s` est positionné pour le fichier de référence `/bin/passwd` contenant le programme réalisant cette commande.

Exemple:

```
                /* fichier test_s.c */

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

main()
{
  FILE * fp;
```

```
     printf("Mon login est : %s\n",getlogin()) ;
     printf("je vais tenter d'ouvrir le fichier
                             /users/obernard/Projet/essai\n");
     if((fp=fopen("/users/obernard/Projet/essai","w")) == NULL )
          perror("Erreur fopen()");
     else
          printf ("le fichier essai est ouvert\n");
}
```

Résultat de l'exécution:

Soit `obernard` **le propriétaire du fichier** `test_s`**. Après avoir créé le fichier** `essai` **à l'aide du programme** `test_s`**, il interdit son accès en écriture aux autres utilisateurs du groupe. Considérons un utilisateur** `elsensoh` **du même groupe que** `obernard`**, qui tente d'ouvrir le fichier** `essai`**, en utilisant le programme** `test_s` **(qui ne lui appartient pas). Evidemment sa tentative échoue, puisqu'il ne possède pas les droits d'accès au fichier** `essai`**:**

```
obernard/> ls -l test_s
-rwxrwxr-x   1 obernard speinf     47551 Jun 11 18:55 test_s
obernard/> test_s
Mon login est : obernard
je vais tenter d'ouvrir le fichier /users/obernard/Projet/essai
le fichier essai est ouvert
obernard/> ls -l essai
-rw-rw-r--   1 obernard speinf         0 Jun 11 18:59 essai
obernard/> chmod g-w essai
obernard/> ls -l essai
-rw-r--r--   1 obernard speinf         0 Jun 11 18:59 essai

elsensoh/> /users/obernard/Projet/test_s
 Mon login est : elsensoh
je vais tenter d'ouvrir le fichier /users/obernard/Projet/essai
Erreur fopen(): Permission denied
```

Maintenant, `obernard` **va positionner le bit** `s` **du programme** `test_s`**, tout en laissant l'interdiction d'accès au fichier** `essai` **aux autres utilisateurs du groupe.**

```
obernard/> chmod u+s test_s
obernard/> ls -l test_s
-rwsrwxr-x   1 obernard speinf     29795 Jun 11 19:03 test_s
elsensoh/> /users/obernard/Projet/test_s
 Mon login est : elsensoh
je vais tenter d'ouvrir le fichier /users/obernard/Projet/essai
le fichier essai est ouvert
```

L'utilisateur `elsensoh` **exécutant le programme** `/users/obernard/Projet/test_s`**, a réussi cette fois à ouvrir le fichier** `essai`**. En effet, grâce au bit** `s`**,** `elsensoh` était transformé en le propriétaire du fichier `test_s` le temps de son exécution.

En résumé, on peut dire qu'un processus a le droit d'accéder à un fichier si:

1. L'identificateur d'utilisateur effectif du processus est l'identificateur du super-utilisateur.
2. L'identificateur d'utilisateur effectif du processus est identique à l'identificateur du propriétaire du fichier, et le mode du fichier comporte le droit désiré dans le champ *propriétaire*.
3. L'identificateur d'utilisateur effectif du processus n'est pas l'identificateur du propriétaire du fichier, et l'identificateur de groupe effectif du processus est identique à l'identificateur de groupe du fichier, et le mode du fichier comporte le droit désiré dans le champ *groupe*.
4. L'identificateur d'utilisateur effectif du processus n'est pas l'identificateur du propriétaire du fichier, et l'identificateur de groupe effectif du processus n'est pas l'identificateur de groupe du fichier, et le mode du fichier comporte le droit désiré dans le champ *autres*.

1.4.3 Identificateurs d'utilisateur

A chaque processus sont associés 2 entiers, l'identificateur d'utilisateur réel et l'identificateur d'utilisateur effectif.
L'identificateur d'utilisateur réel identifie dans tous les cas l'utilisateur exécutant le processus.
L'identificateur d'utilisateur effectif est utilisé pour déterminer les permissions du processus. Ces deux valeurs sont en général égales. En changeant l'identificateur d'utilisateur effectif, un processus gagne les permissions associées au nouvel identificateur d'utilisateur (et perd temporairement celles associées à l'identificateur d'utilisateur réel).

Exemple:

```
                /* fichier test_acces.c */

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

main()
{
FILE *fp ;
     printf("mon login est : %s\n",getlogin()) ;
     printf("je vais tenter d'ouvrir le fichier /users/obernard/essai\n") ;
     if ((fp=fopen("/users/obernard/essai","w")) == NULL)
          perror("Erreur fopen()") ;
     else
```

```
          printf("le fichier /users/obernard/essai est ouvert\n") ;
     printf("je vais tenter d'ouvrir le fichier /users/elsensoh/essai\n") ;
     if ((fp=fopen("/users/elsensoh/essai","w")) == NULL)
          perror("Erreur fopen()") ;
     else
          printf("le fichier /users/elsensoh/essai est ouvert\n") ;
}
```

On a rajouté au programme précédent l'ouverture d'un fichier appartenant à elsensoh. Le bit s est positionné pour test_acces.

```
-rw-r--r--   1 obernard speinf        11 Jun 20 18:15 essai
-rwsrwxr-x   1 obernard speinf     30047 Jun 20 18:02 test_acces

-rw-r--r--   1 elsensoh speinf        11 Jun 20 18:09 essai
```

Maintenant, obernard lance test_acces. On peut constater qu'il a accès au fichier essai qui lui appartient, mais pas à celui de elsensoh:

```
mon login est : obernard
je vais tenter d'ouvrir le fichier /users/obernard/essai
le fichier /users/obernard/essai est ouvert
je vais tenter d'ouvrir le fichier /users/elsensoh/essai
fopen: Permission denied
```

A son tour, elsensoh lance test_acces et il obtient les droits sur le fichier essai appartenant à obernard, mais il perd les droits sur son propre fichier essai :

```
mon login est : elsensoh
je vais tenter d'ouvrir le fichier /users/obernard/essai
le fichier /users/obernard/essai est ouvert
je vais tenter d'ouvrir le fichier /users/elsensoh/essai
fopen: Permission denied
```

Obtention et modification des identificateurs d'utilisateur

Les identificateurs d'utilisateur réel et effectif sont obtenus à l'aide des appels système getuid() et geteuid().

```
int getuid()          /* determine l'identificateur d'utilisateur reel */
int geteuid()         /* determine l'identificateur d'utilisateur effectif */
```

Valeur retournée: identificateur réel ou effectif. Pas d'erreur possible.

Ces identificateurs peuvent être modifiés par l'appel système setuid().

```
int setuid(uid)        /* change les identificateurs d'utilisateur */
int uid ;              /* valeur a affecter */
```

Valeur retournée: -1 en cas d'erreur.

De plus, l'utilisation de ces primitives entaîne la sauvegarde de l'identificateur précédent.

La gestion de `setuid()` est régie par trois règles:

1. Si cet appel système est employé par le super-utilisateur, il positionne l'identificateur d'utilisateur effectif ainsi que l'identificateur d'utilisateur réel à la valeur donnée en argument. Il permet ainsi au super-utilisateur de devenir n'importe quel utilisateur.

   Dans le cas des utilisateurs ordinaires:

2. Si l'identificateur d'utilisateur réel est égal à la valeur passée en argument, alors l'identificateur d'utilisateur effectif prend cette valeur. Cela permet au processus de retrouver les permissions de l'utilisateur l'exécutant, après application de la troisième règle.

3. Si l'identificateur sauvegardé est égal à la valeur passée en argument, l'identificateur d'utilisateur effectif prend cette valeur. Cela permet à un processus de s'exécuter temporairement avec les permissions d'un autre utilisateur. Le programme peut retrouver ses permissions originales en appliquant la deuxième règle.

Tâchons d'éclairer quelque peu notre propos à l'aide de l'exemple suivant:

Le code est écrit et compilé sous le `login` `obernard`, d'identificateurs réel et effectif 4022.
Le programme est lancé successivement depuis les `login` `obernard` puis `elsensoh` (d'identificateurs réel et effectif 4010).

```
                /* fichier test_uid.c */

#include <errno.h>
#include <stdio.h>
main()
{
printf(" mon login est : %s\n",getlogin()) ;
printf("voici mon      real user id : %d\n",getuid()) ;
printf("voici mon effective user id : %d\n",geteuid()) ;
        if (setuid(4010) == -1) {
               perror("Erreur setuid()") ;
               exit(-1) ;
        }
printf("voici mon      real user id : %d\n",getuid()) ;
```

```
printf("voici mon effective user id : %d\n",geteuid()) ;

        if (setuid(4022) == -1) {
               perror("Erreur setuid") ;
               exit(-1) ;
        }
printf("voici mon      real user id : %d\n",getuid()) ;
printf("voici mon effective user id : %d\n",geteuid()) ;
}
```

Résultat de l'exécution:

```
obernard/> ls -l test_uid
-rwxrwxr-x   1 obernard speinf     27938 Jun 11 17:12 test_uid
obernard/> testuid
 mon login est : obernard
voici mon      real user id : 4022
voici mon effective user id : 4022
Erreur setuid(): Not owner
```

Le premier `setuid()` est interrompu, en effet aucune des règles d'application n'est respectée. Essayons de lancer le programme à partir du login `elsensoh`, non propriétaire du fichier:

```
elsensoh/> /users/obernard/test_uid
 mon login est : elsensoh
voici mon      real user id : 4010
voici mon effective user id : 4010
voici mon      real user id : 4010
voici mon effective user id : 4010
setuid: Not owner
```

Le premier `setuid()` s'exécute avec succès : l'identificateur d'utilisateur réel (4010) est égal à la valeur passée en argument. Pour pouvoir exécuter le second `setuid()`, le propriétaire du fichier, `obernard`, doit actionner le bit `s` pour que `elsensoh` soit le temps de l'exécution propriétaire du fichier.

```
obernard/> chmod u+s test_uid
obernard/> ls -l test_uid
-rwsrwxr-x   1 obernard speinf     27938 Jun 11 17:12 test_uid

elsensoh/> /users/obernard/test_uid
 mon login est : elsensoh
voici mon      real user id : 4010
voici mon effective user id : 4022
voici mon      real user id : 4010
voici mon effective user id : 4010
voici mon      real user id : 4010
voici mon effective user id : 4022
```

Avant le lancement du programme, les identificateurs réel et effectif sont égaux (4010). Ensuite, l'identificateur effectif devient 4022 lorsque le programme s'exécute, en raison de la présence du bit s.
Lors de l'appel de `setuid(4010)`, l'identificateur réel est égal à l'argument (deuxième règle), il y a donc sauvegarde de la valeur 4022 dans l'identificateur sauvegardé. Lors de l'appel de `setuid(4022)`, l'identificateur sauvegardé (4022) est égal à l'argument (troisième règle), le processus retrouve son identificateur initial, et donc les permissions associées.

Intérêts de la chose:

1. Gestion du courrier électronique.
2. Gestion des imprimantes.
3. SGBD (possibilité de créer des fichiers appartenant à l'utilisateur et non à la base entière).
4. Login (après avoir demandé un nom d'utilisateur et un mot de passe, `login` les vérifie en consultant `/etc/passwd`, et s'ils sont conformes, elle exécute `setuid()` et `setgid()` pour positionner les identificateurs d'utilisateur et de groupe réel et effectif aux valeurs de l'entrée correspondante de `/etc/passwd`).

1.4.4 Identificateurs de groupe

Le principe est identique à celui des identificateurs d'utilisateur, plusieurs utilisateurs pouvant appartenir au même groupe, leur permettant d'avoir accès aux fichiers du groupe, et refusant cet accès aux autres.

Les appels système pour obtenir les identificateurs de groupe réel et effectif et pour les modifier sont respectivement `getgid()`, `getegid()` et `setgid()`.

Il faut noter que sous System V, un utilisateur ne peut appartenir qu'à un groupe à la fois. Pour changer de groupe, il faut exécuter la commande `newgrp()`, qui change l'identificateur de groupe et exécute un nouveau shell.
Remarque: si le groupe auquel on veut appartenir a installé un mot de passe de groupe (de plus en plus inusité), celui-ci est demandé.

1.5 Entrées/Sorties fichiers

1.5.1 Notion de table de nœuds d'index

Cette table est placée au début de chaque région de disque contenant un système de fichiers UNIX. Chaque nœud d'index de cette table correspond à un fichier et contient des informations essentielles sur les fichiers inscrits sur le disque:

- le type du fichier (détaillé plus bas).

- le nombre de liens (nombre de noms de fichier donnant accès au même fichier).
- le propriétaire et son groupe.
- l'ensemble des droits d'accès associés au fichier pour le propriétaire du fichier, le groupe auquel appartient le propriétaire, et enfin tous les autres usagers.
- la taille en nombre d'octets.
- les dates du dernier accès, de la dernière modification, et du dernier changement d'état (quand le nœud d'index lui-même a été modifié).
- des pointeurs vers les blocs du disque contenant le fichier proprement dit.

La structure `stat` correspondante est définie comme suit, dans le fichier `<sys/stat.h>`:

```
struct    stat {
     dev_t     st_dev;   /* identificateur du peripherique qui contient */
                         /* le repertoire ou se trouve le fichier       */
     ino_t     st_ino;   /* numero du noeud d'index                     */
     mode_t    st_mode;  /* droits d'acces du fichier                   */
     nlink_t   st_nlink; /* nombre de liens effectues sur le fichier    */
     uid_t     st_uid;   /* identificateur du proprietaire              */
     gid_t     st_gid;   /* identificateur du groupe du proprietaire    */
     dev_t     st_rdev;  /* identificateur du peripherique qui contient */
                         /* le fichier                                  */
     off_t     st_size;  /* taille en octets du fichier                 */
     time_t    st_atime; /* date du dernier acces au fichier            */
     time_t    st_mtime; /* date de la derniere modification du fichier */
     time_t    st_ctime; /* date du dernier changement du noeud d'index */
};
```

Remarque:

Cette table ne contient ni le nom du fichier, ni les données.

1.5.2 Les types de fichiers

Il y a trois espèces de fichiers UNIX :

- les fichiers ordinaires: tableaux linéaires d'octets n'ayant pas de nom, identifiés par leur numéro d'index.
- les répertoires: l'utilité de ces fichiers est de permettre de repérer un fichier par un nom plutôt que par son numéro d'index dans la table de nœud d'index; le répertoire est donc constitué d'une table à deux colonnes

contenant d'un coté le nom que l'utilisateur donne au fichier, et de l'autre, le numéro d'index qui permet d'accéder à ce fichier. Cette paire est appelée un lien.

- les fichiers spéciaux : Il s'agit d'un type de périphérique, ou d'un FIFO (file, first in first out).

1.5.3 Descripteurs de fichier – Pointeurs vers fichier

Nous avons vu que le nœud d'index d'un fichier est la structure d'identification du fichier vis à-vis du système. Maintenant, lorsqu'un processus veut manipuler un fichier, il va utiliser plus simplement un entier appelé descripteur de fichier. L'association de ce descripteur au nœud d'index du fichier se fait lors de l'appel à la primitive `open()`(voir en 1.5.4), le descripteur devenant alors le nom local du fichier dans le processus. Chaque processus UNIX dispose de 20 descripteurs de fichier, numérotés de 0 à 19. Par convention, les trois premiers sont toujours ouverts au début de la vie d'un processus:

- Le descripteur de fichier 0 est l'entrée standard (généralement le clavier).
- Le descripteur de fichier 1 est la sortie standard (généralement l'écran).
- Le descripteur de fichier 2 est la sortie erreur standard (généralement l'écran).

Les 17 autres sont disponibles pour les fichiers et les fichiers spéciaux que le processus ouvre lui-même.
Cette notion de descripteur de fichier est utilisée lors de l'interface d'entrée/sortie de bas niveau, notamment avec les primitives `open()`, `write()` ...

En revanche, lorsqu'on utilise les primitives de la librairie standard d'entrées/sorties, les fichiers sont repérés par des pointeurs vers des objets de type `FILE` (type défini dans le fichier `<stdio.h>`).
Il y a trois pointeurs prédéfinis:

- *stdin* qui pointe vers le tampon du standard input (généralement le clavier).
- *stdout* qui pointe vers le tampon du standard output (généralement l'écran).
- *stderr* qui pointe vers le tampon du standard error output (généralement l'écran).

1.5.4 Description de quelques primitives et fonctions

Primitive creat()

```
int creat(path,perm)              /* cree un fichier */
char *nom ;                       /* nom avec le chemin d'acces */
int perm ;                        /* droits d'acces */
```

Valeur retournée: descripteur de fichier ou -1 en cas d'erreur.

Cette primitive réalise la création d'un fichier, dont le nom est donné dans le paramètre *path*. L'entier *perm* correspond au nombre octal (droits d'accès) décrit en 1.4.2.
Si le fichier n'existait pas, il est ouvert en écriture.
Sinon, l'autorisation en écriture n'est pas nécessaire; il suffit d'avoir l'autorisation en exécution. Dans ce cas, l'utilisateur et le groupe effectifs deviennent les propriétaires du fichier. Ni les propriétaires du fichier, ni les autorisations ne sont modifiés, mais sa longueur est tronquée à zéro, et le fichier est ouvert en écriture (l'argument *perm* est alors ignoré).
Pour créer un fichier de nom "`essai_creat`" avec les autorisations lecture et écriture pour le propriétaire et le groupe, on écrira:

```
if ((fd=creat("essai_creat",0666)) == -1)
        perror("Erreur creat()") ;
```

Primitive open()

```
#include <fcntl.h>
#include <sys/types.h>

int open(path,mode,perm)          /* ouvre un fichier */
char *path                        /* chemin d'acces */
int mode                          /* option d'ouverture */
int perm                          /* droits d'acces */
```

Valeur retournée: descripteur de fichier ou -1 en cas d'erreur.

Cette primitive permet d'ouvrir (ou de créer) le fichier de nom *path*. L'entier *mode* détermine le mode d'ouverture du fichier. Le paramètre *perm* n'est utilisé que lorsque `open()` réalise la création du fichier. Dans ce cas, il indique les droits d'accès au fichier.
Le paramètre *mode* peut prendre une ou plusieurs des constantes symboliques (qui sont dans ce cas séparées par des OU), définies dans `<fcntl.h>`:

Drapeaux utilisés avec `open()`		
valeur	**mnémonique**	**fonction**
00000	**O_RDONLY**	**ouverture en lecture seule**
00001	**O_WRONLY**	**ouverture en écriture seule**
00002	**O_RDWR**	**ouverture en lecture et écriture**
00004	**O_NDELAY**	**empêche le blocage d'un processus lors d'un appel en lecture sur un tube, un tube nommé (FIFO) ou un fichier spécial, non encore ouvert en écriture; renvoie une erreur lors d'un appel en écriture sur un tube, un tube nommé ou un fichier spécial, non encore ouvert en lecture. Là encore, le blocage du processus est évité**
00010	**O_APPEND**	**ouverture en écriture à la fin du fichier**
00400	**O_CREAT**	**création du fichier s'il n'existe pas (seul cas d'utilisation de l'argument *perm*)**
01000	**O_TRUNC**	**troncature à la longueur zéro, si le fichier existe**
02000	**O_EXCL**	**si O_CREAT est aussi positionné, provoque un échec si le fichier existe déjà.**

Exemple:

Pour effectuer la création et l'ouverture du fichier "`essai_open`" en écriture avec les autorisations suivantes:
***lecture et écriture pour le propriétaire et le groupe*,**
il faut écrire:

```
if ((fd=open(path,O_WRONLY|O_CREAT|O_TRUNC,0666)) == -1)
        perror("Erreur open()") ;
```

Remarque:

L'inclusion du fichier `<sys/types.h>` est nécessaire, car des types utilisés dans `<fcntl.h>` y sont définis.

Fonction fdopen()

Cette fonction permet de faire le lien entre les manipulations de fichiers de la librairie standard C, qui utilise des pointeurs vers des objets de type `FILE` (`fclose()`, `fflush()`, `fprintf()`, `fscanf()...`), et les primitives de bas niveau (`open()`, `write()`, `read()...`) qui utilisent des descripteurs de fichiers de type `int`. La description de la bibliothèque standard n'étant pas l'objet de notre propos, nous n'irons pas plus avant dans ce domaine.

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

```
FILE* fdopen(fd,mode)             /* convertit un descripteur de fichier en */
                                  /* un pointeur sur un fichier             */
int fd ;                          /* descripteur concerne par la conversion */
char *mode ;                      /* mode d'ouverture desire               */
```

Valeur retournée: un pointeur sur le fichier associé au descripteur `fd`, ou la constante prédéfinie (dans `<stdio.h>`) NULL en cas d'erreur.

Remarque:

Le fichier doit, au préalable, avoir été ouvert à l'aide de la primitive `open()`. D'autre part, le paramètre *mode* choisi doit être compatible avec le mode utilisé lors de l'ouverture par `open`.
Ce paramètre peut prendre les valeurs suivantes:

- "r" : le fichier est ouvert en lecture.
- "w" : le fichier est créé et ouvert en écriture. S'il existait déjà, sa longueur est ramenée à zéro.
- "a" : ouverture en écriture à la fin du fichier (avec création préalable si le fichier n'existait pas).

Exemple:

```
/* ouverture prealable par open() par exemple en lecture */
if ((fd=open("mon_fichier",O_RDONLY,0666)) == -1)
        perror("Erreur open()") ;
/* association de fp (de type FILE*) a fd (de type int)   */
if ((fp=fdopen(fd,"r")) == NULL)
        perror("Erreur fdopen()") ;
```

Primitive close()

```
int close(fd)                     /* fermeture de fichier */
int fd ;                          /* descripteur de fichier */
```

Valeur retournée: 0 ou -1 en cas d'échec.

Cette primitive ne vide pas le tampon du noyau correspondant au processus, elle libère simplement le descripteur de fichier pour une éventuelle réutilisation.

Primitives dup() - dup2()

```
int dup(fd)             /* duplication d'un descripteur de fichier */
int fd ;                /* descripteur a dupliquer */
```

Valeur retournée: nouveau descripteur de fichier ou -1 en cas d'erreur.

Cette primitive duplique un descripteur de fichier existant et fournit donc un descripteur ayant exactement les mêmes caractéristiques que celui passé en argument. Elle garantit que la valeur retournée est la plus petite possible parmi les valeurs de descripteurs disponibles.

```
int dup2(fd1,fd2)     /* force fd2 comme synonyme de fd1 */
int fd1 ;             /* descripteur a dupliquer */
int fd2 ;             /* nouveau descripteur */
```

Valeur retournée: -1 en cas d'erreur.

Cette primitive a la même fonction que `dup()`**. Notons que** `dup2()` **ferme le descripteur fd2 si celui-ci était ouvert.**

Exemple:

```
                /* fichier test_dup.c */

#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>

main()
{
 int fd ;               /* descripteur a dupliquer  */
 int retour1=10 ;       /* valeur de retour de dup  */
 int retour2=10 ;       /* valeur de retour de dup2 */

     if ((fd=open("./fic",O_RDWR|O_CREAT|O_TRUNC,0666))==-1) {
          perror("Erreur open()") ;
          exit(1) ;
     }
     close(0) ;         /* fermeture du standard stdin */

     if ((retour1 = dup(fd)) == -1) {  /* duplication */
          perror("Erreur dup()") ;
          exit(1) ;
     }

     if ((retour2 = dup2(fd,1)) == -1) {  /* duplication de stdout */
          perror("Erreur dup2()") ;
          exit(1) ;
     }

     printf ("valeur de retour de dup()  : %d \n",retour1) ;
```

```
    printf ("valeur de retour de dup2() : %d \n",retour2) ;
}
```

Résultat de l'exécution:

```
bolz> test_dup
bolz>
bolz> cat fic
valeur de retour de dup()  : 0
valeur de retour de dup2() : 1
bolz>
```

L'appel à `dup()` redirige l'entrée standard vers le fichier `fic`, de descripteur *fd*, et l'appel à `dup2()` redirige la sortie standard vers ce même fichier. Le résultat de l'exécution n'est donc pas visualisé sur l'écran, il faut éditer `fic`. On remarque que l'appel à `dup2()` ne nécessite pas la fermeture du descripteur à dupliquer.

Primitive write()

```
int write(fd, buf, nbytes)          /* ecriture dans un fichier */
int fd ;                            /* descripteur de fichier */
char *buf ;                         /* adresse du tampon */
unsigned nbytes ;                   /* nombre d'octets a ecrire */
```

Valeur retournée: nombre d'octets écrits ou -1 en cas d'erreur.

Cette primitive écrit dans le fichier ouvert représenté par *fd* les *nbytes* octets sur lesquels pointe *buf*. Il faut noter que l'écriture ne se fait pas directement dans le fichier, mais passe par un tampon du noyau (méthode du "kernel buffering").

Primitive read()

```
int read(fd, buf, nbytes)           /* lecture dans un fichier */
int fd ;                            /* descripteur de fichier */
char *buf ;                         /* adresse du tampon */
unsigned nbytes ;                   /* nombre d'octets a lire */
```

Valeur retournée: nombre d'octets lus, 0 sur EOF, ou -1 en cas d'erreur.

La primitive lit les *nbytes* octets dans le fichier ouvert représenté par *fd*, et les place dans le tampon sur lequel pointe *buf*.

Remarque:

Les opérations d'ouverture de fichiers (semblables à `open()`), et de duplication de descripteurs (semblables à `dup()`) ont été réunies dans la primitive `fcntl()`, qui ne sera pas détaillée ici (voir `/usr/include/fcntl.h`).

Exemple 1: Redirection standard.

Ce programme exécute la commande shell `ps`, puis redirige le résultat vers un fichier `fic_sortie`. Ainsi l'exécution de ce programme ne devrait plus rien donner à l'écran. La primitive `execl()` exécute la commande passée en argument, nous nous attarderons sur cette primitive dans le chapitre suivant, concernant les processus.

```
                /* fichier test_dup2.c */

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

#define STDOUT 1

main()
{
 int fd ;
     /* affecte au fichier fic_sortie le descripteur fd */
     if ((fd = open("fic_sortie",O_CREAT|O_WRONLY|O_TRUNC,0666)) == -1){
          perror("Erreur sur l'ouverture de fic_sortie") ;
          exit(1) ;
     }
     dup2(fd,STDOUT) ;            /* duplique la sortie standard */
     execl("/bin/ps","ps",NULL) ; /* execute la commande */
}
```

Résultat de l'exécution:

```
obernard/> test_dup2
obernard/> more fic_sortie
   PID TTY      TIME COMMAND
  3954 ttyp3    0:03 ksh
```

Notons que les autres redirections suivent le même principe, et qu'il est possible de coupler les redirections d'entrée et de sortie.

Exemple 2:

Le programme ci-dessous réalise la copie d'un fichier vers un autre. Il est comparable à la commande shell `cp`.

```
                /* fichier copie_fic.c */

#include <errno.h>
#include <sys/types.h>
#include <fcntl.h>
#define TAILLEBUF 512

void copie_fic(src,dest)   /* copie le fichier src */
char *src, *dest ;         /* vers le fichier dest */
{
 int srcfd,destfd ;
 int nlect, necrit, n ;
 char buf[TAILLEBUF] ;

      if ((srcfd = open(src,O_RDONLY)) == -1){      /* ouverture de src */
           perror("ouverture du fichier source") ;  /* en lecture seule */
           exit(1) ;
      }

      if ((destfd = creat(dest,0666)) == -1){       /* creation de dest */
           perror("creation du fichier destination");
           exit(1) ;
      }
      while ((nlect = read(srcfd,buf,sizeof(buf))) != 0){  /* lecture  */
           if (nlect == -1){                               /* dans src */
                perror("Erreur read()") ;
                exit(1) ;
           }
           necrit = 0 ;
           do {                       /* ecriture dans dest */
                if ((n = write(destfd,&buf[necrit],nlect-necrit)) == -1)
                {
                     perror("Erreur write()") ;
                     exit(1) ;
                }
                necrit += n ;
           } while (necrit < nlect) ;

      }
      if (close(srcfd) == -1 || close(destfd) == -1){
           perror("Erreur close()") ;
           exit(1) ;
      }
}

main(argc,argv)
int argc ;
```

```
char *argv[] ;
{
     if (argc != 3) {          /* verifie le nombre d'arguments */
          printf("Usage: copie_fic fichier1 fichier2\n") ;
          exit(1) ;
     }
     printf("Je suis en train de copier ...\n") ;
     copie_fic(argv[1],argv[2]) ;
     printf("J'ai fini!\n") ;
}
```

Résultat de l'exécution:

```
bolz> cat>fic1
voici le contenu du fichier a copier.
bolz> copie_fic fic1 fic2
Je suis en train de copier ...
J'ai fini!
bolz> cat fic2
voici le contenu du fichier a copier.
bolz>
```

1.6 Clé d'accès

Une clé est un entier long. Elle est utilisée pour identifier une structure de données à laquelle un programme veut se référer. Il existe une fonction qui permet de créer des clés, et ce de manière unique: `ftok()`.

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path,id)                /* renvoie une cle */
char *path ;                       /* nom d'un fichier existant */
int id ;
```

Valeur retournée: valeur d'une clé unique pour tout le système ou -1 en cas d'erreur. `ftok()` utilise le nom du fichier (*path*), qui est unique dans le système, comme une chaîne de caractères, et la combine à *id* pour générer une valeur de clé.

Il est également possible de créer des fonctions générant des clés en s'appuyant sur des paramètres liés à l'utilisateur, comme son numéro d'identificateur et son numéro de groupe:

```
#define CLE(n)  ((getuid() % 100) * 100 + getgid() + n)
```

Chapter 2

LES PROCESSUS

Définition

Un processus est un environnement d'exécution qui consiste en un segment d'instructions, et deux segments de données (data et stack). Il faut bien faire la différence avec la notion de programme qui n'est autre qu'un fichier contenant des instructions et des données utilisées pour initialiser les segments d'instructions et de données de l'utilisateur d'un processus.

2.1 Les identificateurs de processus

Chaque processus possède un identificateur unique nommé ***pid***. Comme pour les utilisateurs, il peut être lié à un groupe; on utilisera alors l'identificateur ***pgrp***. Citons les différentes primitives permettant de connaître ces différents identificateurs:

```
int getpid()       /* retourne l'identificateur du processus */

int getpgrp()      /* retourne l'identificateur du groupe de processus */

int getppid()      /* retourne l'identificateur du pere du processus */

int setpgrp()      /* positionne  l'identificateur du groupe de */
                   /* processus a la valeur de son pid: cree un */
                   /* nouveau groupe de processus              */
```

Valeur retournée: nouvel identificateur du groupe de processus.

Exemple:

```
                 /* fichier test_idf.c */

main()
{
     printf("je suis le processus %d de pere %d et de groupe %d\n",getpid()
```

```
             ,getppid(),getpgrp()) ;
}
```

Résultat de l'exécution:

```
obernard/> ps
   PID TTY      TIME COMMAND
  6658 ttyp5    0:04 ksh
obernard/> test_idf
je suis le processus 8262 de pere 6658 et de groupe 8262
```

Notons que le père du processus exécutant `test_idf` est ksh.

Remarque:

Il semblerait que `ksh` et `csh` utilisent la notion de groupe de processus d'une façon particulière. En effet, le programme ci-dessous provoque un break total (utilisateur délogué) dans `ksh` et `csh` mais pas dans `sh`.

```
                 /* fichier test_pgid.c */

/*
 * ce programme, qui devrait creer un groupe de processus,
 * delogue l'utilisateur, lorsqu'il est execute sous csh et ksh
 */

main()
{
     printf("je cree le groupe de pgid = %d\n",setpgrp()) ;
}
```

2.2 Les primitives principales de gestion de processus

2.2.1 Primitive fork()

```
int fork()          /* creation d'un fils */
```

Valeur retournée: 0 pour le processus fils, et l'identificateur du processus fils pour le père, -1 dans le cas d'épuisement de ressource.

Cette primitive est l'unique appel système permettant de créer un processus. Les processus père et fils partagent le même code. Le segment de données de l'utilisateur du nouveau processus est une copie exacte du segment correspondant de l'ancien processus, alors que la copie du segment de données système diffère par certains attributs (par exemple le *pid*, le temps d'exécution ...).
Le fils hérite d'un double de tous les descripteurs de fichiers ouverts du père

(si le fils en ferme un, la copie du père n'est pas changée), par contre les pointeurs de fichier associés sont partagés (si le fils se déplace dans le fichier, la prochaine manipulation du père se fera à la nouvelle adresse). Cette notion est importante pour implémenter des tubes.

Exemple:

```
                 /* fichier test_fork1.c */

/*
 * heritage du fils en ce qui concerne les descripteurs
 */

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

main()
{
int pid ;
int fd ;       /* descripteur de fichier qu'on associe au fichier calepin */
char *telephone ;
int r ;        /* retour de read */
int i ;
char c ;
     printf("bonjour, je suis %d\n",getpid()) ;
     printf("donnez-moi votre numero de telephone\n") ;
     printf("c'est le 123456789 ? Ok, je l'inscris dans mon calepin\n") ;
     if((fd=open("calepin",O_CREAT|O_RDWR|O_TRUNC,775))==-1)
     {
          perror("impossible d'ouvrir calepin") ;
          exit(-1) ;
     }
     telephone="123456789" ;
     if(write(fd,telephone,9)==-1)
     {
          perror("impossible d'ecrire dans calepin") ;
          exit(-1) ;
     }
     printf("voila, j'ai fini de le noter, je referme le calepin\n") ;
     close(fd) ;
     printf("quoi ? je me suis trompe ? qu'avais-je note au fait ?\n") ;
     printf("\tle pere reouvre son calepin\n") ;
     if((fd=open("calepin",O_RDONLY,775))==-1)
     {
          perror("impossible d'ouvrir calepin") ;
```

```
        exit(-1) ;
    }
    printf("\ta cet instant, le pere enfante un fils\n") ;
    pid=fork() ;
    if(pid==-1) /* erreur */
    {
        perror("impossible de creer un fils") ;
        exit(-1) ;
    }
    else if(pid==0) /* fils */
    {
        sleep(1) ; /* on endort le fils pour grouper les messages */
        printf("\t\tsalut, c'est moi %d\n",getpid()) ;
        printf("\t\tvous savez, moi aussi je sais lire\n") ;
        printf("\tle fils se jette alors sur le calepin\n") ;
        for(i=1;i<=5;i++)
        {
            if(read(fd,&c,1)==-1)
            {
                perror("impossible de lire calepin") ;
                exit(-1) ;
            }
            printf("\t\tje lis un %c\n",c) ;
        }
        printf("\tmon calepin ! dit le pere\n") ;
        printf("\tet de surprise le fils referma le calepin\n") ;
        close(fd) ;
        sleep(3) ;
        printf("\tle fils se suicida alors de desespoir\n") ;
        exit(1) ;
    }
    else /* pere */
    {
        printf("au fait, je vous presente mon fils %d\n",pid) ;
        sleep(2) ;
        printf("galopin ! j'ai plus d'un tour dans mon sac :\n");
        printf("je peux lire mon calepin quand meme\n") ;
        while((r=read(fd,&c,1))!=0)
        {

            if(r==-1)
            {
                perror("impossible de lire calepin") ;
                exit(-1) ;
            }
            printf("%c",c) ;
        }
```

```
          printf("\n") ;
          printf("M'ENFIN ! ou est passe le reste ?\n") ;
          sleep(3) ;
          close(fd) ;
     }
}
```

Résultat de l'exécution:

```
bonjour, je suis 10349
donnez-moi votre numero de telephone
c'est le 123456789 ? Ok, je l'inscris dans mon calepin
voila, j'ai fini de le noter, je referme le calepin
quoi ? je me suis trompe ? qu'avais-je note au fait ?
        le pere reouvre son calepin
        a cet instant, le pere enfante un fils
                salut, c'est moi 10350
                vous savez, moi aussi je sais lire
        le fils se jette alors sur le calepin
                je lis un 1
                je lis un 2
                je lis un 3
                je lis un 4
                je lis un 5
        mon calepin ! dit le pere
        et de surprise le fils referma le calepin
galopin ! j'ai plus d'un tour dans mon sac :
je peux lire mon calepin quand meme
6789
M'ENFIN ! ou est passe le reste ?
        le fils se suicida alors de desespoir
```

Remarque:

Dans cette exemple, on peut voir que:

1. **le fils hérite des descripteurs ouverts du père (puisqu'il peut lire dans calepin sans l'ouvrir).**
2. **si le fils ferme un descripteur ouvert par le père, ce descripteur reste ouvert pour le père.**
3. **les 2 processus partagent le même pointeur sur le fichier de descripteur dupliqué au moment du fork !!! (d'où, lorsque le père lit à son tour dans le fichier, il s'est déplacé, involontairement, autant que le fils).**

Comportement vis-à-vis de la console:

Notons que si le père et le fils vivent, un interruption clavier les tuera tous les deux. En revanche, si un fils vit alors que son père est mort, une interruption de la console ne le tue pas.

```
                 /* fichier test_fork2.c */

/*
 * test des reactions a la console, d'un fils dont le pere est mort
 */

#include <errno.h>
#include <signal.h>

main()
{
int pid ;
     printf("je suis le pere %d et je vais enfanter\n",getpid()) ;
     pid=fork() ; /* creation du fils */
     if(pid==-1) /* erreur */
     {
          perror("impossible de creer un fils\n") ;
     }
     else if(pid==0) /* actions du fils */
     {
          printf("\tbonjour, je suis %d le fils\n",getpid()) ;
          printf("\tit's a really nice world, isn't it ?\n") ;
          printf("\tje m'y installe definitivement (enfin, je l'espere) !\n") ;
          for(;;) ; /* le fils boucle a l'infini */
     }
     else /* actions du pere */
     {
          sleep(1) ; /* pour separer les affichages du pere et du fils */
          printf("il est temps que je m'eteigne, moi, %d\n",getpid()) ;
          printf("mon heure est enfin venue : adieu, %d, mon fils\n",pid) ;
          /* le pere decede de mort naturelle */
     }
}
```

Résultat de l'exécution:

```
je suis le pere 7684 et je vais enfanter
        bonjour, je suis 7685 le fils
        it's a really nice world, isn't it ?
        je m'y installe definitivement (enfin, je l'espere) !
il est temps que je m'eteigne, moi, 7684
mon heure est enfin venue : adieu, 7685, mon fils
```

Maintenant, si on exécute la commande shell ps on obtient:

```
   PID TTY      TIME COMMAND
  6712 ttyp9    0:04 ksh
  7685 ttyp9    0:09 test_for
```

Notre fils tourne toujours!
Essayons de l'interrompre à partir de la console : CTRL-c, CTRL-d ou autre ...
Il est effectivement accroché à la vie.
Maintenant, envoyons-lui un signal direct à l'aide de la commande shell:

```
kill -9 <pid>
```

Dans le cas présent, on ferait:

```
kill -9 7685
```

Cette fois, ça y est, on a réussi à le tuer.

Problème des tampons de sortie:

```
          /* fichier test_fork3.c */

/*
 * le fils herite d'une recopie du tampon de sortie du pere
 */

#include <stdio.h>

main()
{
int pid ;
     printf(" 1") ;
     pid=fork() ;
     if(pid==-1) /* erreur */
     {
          perror("impossible de creer un fils") ;
          exit(-1) ;
     }
     else if(pid==0) /* fils */
     {
          printf(" 2") ;
          exit(0) ;
     }
     else /* pere */
     {
          wait(0) ; /* le pere attend la mort de son fils */
          printf(" 3") ;
     }
}
```

Résultat de l'exécution:

Contrairement à ce qu'on pourrait penser, le résultat n'est pas

```
 1 2 3
```

mais

```
 1 2 1 3
```

En effet, à sa naissance, le fils a hérité du " 1" qui se trouvait dans le tampon de sortie du père. Puis, à sa mort, le tampon de sortie du fils est vidé, on obtient donc l'affichage suivant: `1 2`. Enfin, le père termine son exécution et affiche à son tour: `1 3`.

2.2.2 Primitive wait()

```
int wait(status)        /* en attente de la mort d'un fils */
int *status             /* statut decrivant la mort du fils */
```

Valeur retournée: identificateur du processus mort ou -1 en cas d'erreur.

Le code retourné via *statusp* indique la raison de la mort du processus, qui est:

- soit l'appel de `exit()`, et dans ce cas, l'octet de droite de *statusp* vaut 0, et l'octet de gauche est le paramètre passé à `exit` par le fils.
- soit la réception d'un signal fatal, et dans ce cas, l'octet de droite est non nul. Les 7 premiers bits de cet octet contiennent le numéro du signal qui a tué le fils. De plus, si le bit de gauche de cet octet est 1, un fichier image a été produit (qui correspond au fichier `core` du répertoire courant). Ce fichier image est une copie de l'espace mémoire du processus, et est utilisé lors du débogage.

Exemple:

```
                 /* fichier test_wait.c */

#include <errno.h>
#include <signal.h>

main()
{
     int pid ;
     printf("Bonjour, je me presente, je suis %d.\n",getpid()) ;
     printf("Je sens quelque chose monter en moi... un fils peut-etre!\n") ;

     if (fork() == 0) {         /* creation du premier fils */
          printf("\tSalut, je suis %d, le gamin de %d.\n",getpid(),getppid()) ;
          sleep(3) ;
          printf("\tJe suis si jeune, et deja je sens que je m'affaiblis!\n") ;
          printf("\tCa y est, je passe de vie a trepas!\n") ;
          exit(7) ;             /* on tue le fils */
```

```
    }

    else {
         int ret1, status1 ;
         printf("Attendons que ce morveux disparaisse.\n") ;
         ret1 = wait(&status1) ;   /* attente de la mort du fils */
         if ((status1&255) == 0) {
              printf("Valeur de retour de wait(): %d\n",ret1) ;
              printf("Parametre de exit(): %d\n",(status1>>8)) ; /* decalage */
              printf("Un simple exit a eu raison du petit.\n") ; /* de 8 bits*/
         }
         else
              printf("Mon fils n'est pas mort par exit.\n") ;
         printf("\nC'est toujours moi, %d, et je sens que ca recommence.\n",
                getpid()) ;

         if ((pid=fork()) == 0) {   /* creation du deuxieme fils */
              printf("\tHello, je suis %d, le deuxieme fils de %d\n",
                     getpid(),getppid()) ;
              sleep(3) ;
              printf("\tJe ne tiens pas a suivre l'exemple de mon frere.\n") ;
              printf("\tEt pour cela, je boucle a l'infini!\n") ;
              for(;;) ;
         }
         else {
              int ret2, status2, s ;
              printf("Celui-ci aussi doit mourir.\n") ;
              ret2 = wait(&status2) ; /* attente de la mort du deuxieme fils */
              if ((status2&255) == 0) {
                   printf("Le fils n'est tue pas par un signal\n") ;
              }
              else {
                   printf("Valeur de retour de wait(): %d\n",ret2) ;
                   s = status2&255 ;
                   if (s>127) {
                        printf("Le signal qui a tue mon fils est: %d\n",
                               s-128) ;
                        printf("Il a cree un fichier 'core'\n") ;
                   }
                   else {
                        printf("Le signal qui a tue mon fils est: %d.\n",s) ;
                        printf("Il n'a pas cree de fichier 'core'\n") ;
                   }
              }
         }
    }
}
```

Résultat de l'exécution:

On lance le programme en arrière-plan, et lorsque le deuxième fils boucle à l'infini, on lui envoie un signal par la commande `shell`
`kill -numero_signal pid_fils2`.

```
bolz> test_wait&
[1]     10299
Bonjour, je me presente, je suis 10299.
bolz> Je sens quelque chose monter en moi... un fils peut-etre!
        Salut, je suis 10300, le gamin de 10299.
Attendons que ce morveux disparaisse.
        Je suis si jeune, et deja je sens que je m'affaiblis!
        Ca y est, je passe de vie a trepas!
Valeur de retour de exit(): 7
Un simple exit a eu raison du petit.

C'est toujours moi,10299, et je sens que ca recommence.
        Hello, je suis 10301, le deuxieme fils de 10299
Celui-ci aussi doit mourir.
        Je ne tiens pas a suivre l'exemple de mon frere.
        Et pour cela, je boucle a l'infini!
kill -8 10301
bolz> Valeur de retour de wait(): 10301
Le signal qui a tue mon fils est: 8.
Il a cree un fichier 'core'

[1] +  Done (29)              test_wait&
bolz> ls
core        test_wait    test_wait.c
bolz>
```

Commentaires:

Après la création de chaque fils, le père se met en attente de la mort de celui-ci.
Le premier fils meurt par un appel à `exit()`, le paramètre de `wait()` contient, dans son octet de gauche, le paramètre passé à `exit()` (ici 7).
Le deuxième fils meurt à la réception d'un signal, le paramètre de `wait()` contient, dans ses 7 premiers bits, le numéro du signal (ici 8), et le huitième bit indique si un fichier `'core'` est créé (ici, c'est le cas).

2.2.3 Remarque concernant les processus zombi

Un processus peut se terminer quand son père n'est pas en train de l'attendre. Un tel processus devient un processus zombi. Il est alors identifié par le nom `<defunct>`. Ses segments d'instructions, de données de l'utilisateur et de données système sont supprimés, mais il continue d'occuper une place dans la table des processus du noyau. Lorsqu'il est attendu à son tour, il disparaît.

Exemple:

```
                /* fichier test_defunct.c */

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

main()
{
int pid ;
     printf("je suis %d et je vais creer un fils\n",getpid()) ;
     printf("je bouclerai ensuite a l'infini\n") ;
     pid = fork() ;
     if(pid == -1) /* erreur */
     {
          perror("impossible de creer un fils") ;
          exit(-1) ;
     }
     else if(pid == 0) /* fils */
     {
          printf("je suis %d le fils et je vis\n",getpid()) ;
          sleep(10) ;
          printf("je me suicide par honnetete de conscience\n") ;
          exit(0) ;
     }
     else /* pere */
     {
          for(;;) ; /* le pere boucle a l'infini */
     }
}
```

Résultat de l'exécution:

On lance le programme `test_defunct` en background.

```
je suis 28339 et je vais creer un fils
je bouclerai ensuite a l'infini
je suis 28340 le fils et je vis
je me suicide par honnetete de conscience
```

Le processus 28339 crée un fils 28340, on effectue tout de suite dans le shell un ps. On obtient:

```
  PID TTY      TIME COMMAND
28339 ttyp9    0:08 test_defunct
```

```
28340 ttyp9    0:00 test_defunct
28341 ttyp9    0:00 ps
```

On refait un ps dès que le fils a annoncé sa mort, et on obtient:

```
  PID TTY      TIME COMMAND
28340 ttyp9    0:00 <defunct>
28339 ttyp9    0:16 test_defunct
28341 ttyp9    0:00 ps
```

2.2.4 Primitive exit()

```
void exit(status)       /* terminaison du processus */
int status;             /* status de sortie */
```

Valeur retournée: c'est la seule primitive qui ne retourne jamais.

Tous les descripteurs de fichier ouverts sont fermés. Si le père meurt avant ses fils, le père du processus fils devient le processus init de ***pid*** 1.

Par convention, un code de retour égal à zéro signifie que le processus s'est terminé normalement, et un code non nul (généralement 1 ou -1) signifie qu'une erreur s'est produite.

2.2.5 Les primitives exec()

Il s'agit d'une famille de primitives permettant le lancement de l'exécution d'un programme externe. Il n'y a pas création d'un nouveau processus, mais simplement changement de programme.
Il y a six primitives exec() que l'on peut répartir dans deux groupes: les execl(), pour lesquels le nombre des arguments du programme lancé est connu, puis les execv() où il ne l'est pas. En outre toutes ces primitives se distinguent par le type et le nombre de paramètres passés.

Premier groupe d'exec(): les arguments sont passés sous forme de liste

```
int execl(path,arg0,arg1,...,argn,NULL) /*execute un programme*/
char *path ;     /* chemin du fichier programme */
char *arg0 ;     /* premier argument */
...
char *argn ;     /* (n+1)ieme argument */

int execle(path,arg0,arg1,...,argn,NULL,envp)
char *path ;
char *arg0 ;
...
char *argn ;
char *envp[] ;     /* pointeur sur l'environnement */
```

```
int execlp(file,arg0,arg1,...,argn,NULL)
char *file ;
char *arg0 ;
...
char *argn ;
```

Second groupe d'exec(): les arguments sont passés sous forme de tableau

```
int execv(path,argv)
char  *path ;
char  *argv[] ;     /* pointeur vers le tableau contenant les arguments */

int execve(path,argv,envp)
char  *path ;
char  *argv[] ;
char *envp[] ;

int execvp(file,argv)
char  *file ;
char  *argv[] ;
```

Recouvrement

Lors de l'appel d'une primitive `exec()`, il y a recouvrement du segment d'instructions du processus appelant, ce qui implique qu'il n'y a pas de retour d'un `exec()` réussi (l'adresse de retour a disparu). Autrement dit, le processus appelant la primitive `exec()` meurt.
Le code du processus appelant est détruit, c'est pourquoi l'utilisation des primitives `exec()` sans `fork()` est sans grande utilité.

Exemple:

```
                /* fichier test_exec.c */

#include <stdio.h>

main()
{
      execl("/bin/ls","ls",NULL) ;
      printf ("je ne suis pas mort\n") ;
}
```

Résultat de l'exécution:

```
obernard\> test_exec
fichier1
fichier2
```

```
test_exec
test_exec.c
obernard\>
```

On note que la commande `ls` est réalisée, contrairement à l'appel à `printf()`, ce qui montre que le processus ne retourne pas après `exec()`.
D'où l'intérêt de l'utilisation de la primitive `fork()`:

```
                /* fichier test_exec_fork.c */

#include <stdio.h>

main()
{
     if ( fork()==0 ) execl( "/bin/ls","ls",NULL) ;
     else {
          sleep(2) ;  /* attend la fin de ls pour executer printf() */
          printf ("je suis le pere et je peux continuer\n") ;
     }
}
```

Résultat de l'exécution:

```
obernard/> test_exec_fork
fichier1
fichier2
test_exec
test_exec.c
test_exec_fork
test_exec_fork.c
je suis le pere et je peux continuer
obernard\>
```

Dans ce cas, le fils meurt après l'exécution de `ls`, et le père continue à vivre et exécute `printf()`.

Comportement vis-à-vis des fichiers ouverts

En principe, les descripteurs de fichiers ouverts avant l'appel d'un `exec()` le restent, sauf demande contraire (par la primitive `fcntl()`). L'un des effets du recouvrement est l'écrasement du tampon associé au fichier dans la zone utilisateur, et donc la perte des informations qu'elle contenait. Il est donc nécessaire de forcer avant l'appel à `exec()` le vidage de ce tampon au moyen de la fonction `fflush()`.

Exemple:

```
                 /* fichier test_tampon1.c */
```

```
#include <stdio.h>

main()
{
     printf("vous ne pouvez pas voir ce texte") ;
     execl("/bin/ls","ls",NULL) ;
}
```

Résultat de l'exécution:

```
res
test_tampon1
test_tampon1.c
test_tampon2
test_tampon2.c
test_tampon3
test_tampon3.c
```

On ne voit pas le message, le tampon de sortie ne s'est pas vidé avant d'être écrasé lors du `execl()`.

Maintenant, on utilise `\n`.

Exemple:

```
                 /* fichier test_tampon2.c */
#include <stdio.h>

main()
{
     printf("vous ne pouvez pas voir ce texte\n") ;
     execl("/bin/ls","ls",NULL) ;
}
```

Résultat de l'exécution:

```
vous ne pouvez pas voir ce texte
test_tampon1
test_tampon1.c
test_tampon2
test_tampon2.c
test_tampon3
test_tampon3.c
```

Le message est affiché, `\n` vide le tampon de sortie et retourne à la ligne.

Enfin, dernière opération, on utilise `fflush()` pour vider le tampon de sortie.

Exemple:

```
                /* fichier test_tampon3.c */
#include <stdio.h>

main()
{
     printf("vous ne pouvez pas voir ce texte") ;
     fflush(stdout) ;
     execl("/bin/ls","ls",NULL) ;
}
```

Résultat de l'exécution:

```
vous ne pouvez pas voir ce textetest_tampon.txt
test_tampon1
test_tampon1.c
test_tampon2
test_tampon2.c
test_tampon3
test_tampon3.c
```

Remarques:

- `stdout` est défini dans `<stdio.h>` et correspond à la sortie écran.
- Notons que les commandes internes du shell ne seront pas exécutées par `exec()`. C'est le cas de la commande `cd` qui, si elle était exécutée par un processus fils, serait sans effet parce qu'un attribut changé par un processus fils (ici le répertoire courant) n'est pas remonté au père.
  Pour suivre un chemin relatif (commençant sur le répertoire courant) le noyau doit savoir où commencer. Pour cela, il conserve tout simplement pour chaque processus le numéro d'index du répertoire courant.

Exemple:

Ce programme utilise la primitive `chdir()`. Cette primitive est utilisée essentiellement dans l'implémentation de la commande shell `cd`. Elle change le répertoire courant du processus qui l'exécute.

```
     int chdir(path)     /* change le repertoire courant */
     char *path ;        /* chaine specifiant le nouveau repertoire */
```

```
               /* fichier test_cd.c */

/* le changement de repertoire n'est valable */
/* que le temps de l'execution du processus */

#include <stdio.h>

main()
```

```
{
     if(chdir("..")==-1) /* on va au repertoire precedent */
     {
          perror("impossible de trouver le repertoire specifie") ;
          exit(-1) ;
     }

     /* on execute un pwd qui va tuer le processus et  */
     /* renvoyer le repertoire dans lequel il se trouve */
     if(execl("/bin/pwd","pwd",NULL)==-1)
     {
          perror("impossible d'executer pwd") ;
          exit(-1) ;
     }
}
```

Résultat de l'exécution:

On utilise, une première fois, la commande shell `pwd`. on obtient:

```
/users/lgi/systemeV/bolz/Documentation/Exemples/Processus
```

On lance maintenant `test_cd`:

```
/users/lgi/systemeV/bolz/Documentation/Exemples
```

On relance `pwd`:

```
/users/lgi/systemeV/bolz/Documentation/Exemples/Processus
```

Comme le montre cet exemple, le processus crée par le shell pour l'exécution du programme `test_cd` a hérité du répertoire courant. L'exécution de `chdir("..")` a permis effectivement de remonter dans l'arborescence. Cependant, ce changement n'est valable que pour le processus effectuant `chdir()`. Aussi, à la mort du processus, peut-on constater que le répertoire, dans lequel on se trouve, finalement, est le même qu'au départ.

Comportement vis-à-vis des signaux

Lors de l'appel `exec()`, les signaux (que nous verrons au chapitre 3) ignorés par le processus appelant restent ignorés dans le nouveau programme, et pour tous les autres, le comportement par défaut (interruption du processus) est réadopté.

Part II

COMMUNICATION INTER-PROCESSUS DE BASE

Chapter 3

LES SIGNAUX

3.1 Les signaux gérés par le système

3.1.1 Introduction

Un signal est une interruption logicielle qui est envoyée aux processus par le système pour les informer sur des événements anormaux se déroulant dans leur environnement (violation mémoire, erreur dans les entrées/sorties). Il permet également aux processus de communiquer entre eux.
Un signal peut (à une exception près: SIGKILL) être traité de trois manières différentes:

- Il peut être ignoré. Par exemple, le programme peut ignorer les interruptions clavier générées par utilisateur (c'est ce qui se passe lorsqu'un processus est lancé en background).
- Il peut être pris en compte. Dans ce cas, à la réception d'un signal, l'exécution d'un processus est détournée vers une procédure spécifiée par l'utilisateur, puis reprend où elle a été interrompue.
- Son comportement par défaut peut être restitué par un processus après réception de ce signal.

3.1.2 Types de signaux

Les signaux sont identifiés par le système par un nombre entier. Le fichier `/usr/include/signal.h` contient la liste des signaux accessibles. Chaque signal est caractérisé par un mnémonique.
La liste des signaux usuels est donnée ci-dessous:

- **SIGHUP (1) Coupure: signal émis aux processus associés à un terminal lorsque celui-ci se déconnecte. Il est aussi émis à chaque processus d'un groupe dont le chef se termine.**
- **SIGINT (2) Interruption: signal émis aux processus du terminal lorsqu'on frappe la touche d'interruption (INTR ou CTRL-C) de son clavier.**
- **SIGQUIT (3)* Abandon: idem avec la touche d'abandon (QUIT ou CTRL-D).**
- **SIGILL (4)* Instruction illégale: émis à la détection d'une instruction illégale, au niveau matérielle (exemple: lorsqu'un processus exécute une instruction flottante alors que l'ordinateur ne possède pas d'instructions flottantes câblées).**
- **SIGTRAP (5)* Piège de traçage: émis après chaque instruction en cas de traçage de processus (utilisation de la primitive `ptrace()`).**
- **SIGIOT (6)* Piège d'instruction d'E/S: émis en cas de problème matériel.**
- **SIGEMT (7) Piège d'instruction émulateur: émis en cas d'erreur matérielle dépendant de l'implémentation.**
- **SIGFPE (8)* émis en cas d'erreur de calcul flottant, comme un nombre en virgule flottante de format illégal. Indique presque toujours une erreur de programmation.**
- **SIGKILL (9) Destruction: arme absolue pour tuer les processus. Ne peut être ni ignoré, ni intercepté (voir SIGTERM pour une mort plus douce).**
- **SIGBUS (10)* émis en cas d'erreur sur le bus.**
- **SIGSEGV (11)* émis en cas de violation de la segmentation: tentative d'accès à une donnée en dehors du domaine d'adressage du processus actif.**
- **SIGSYS (12)* Argument incorrect d'un appel système.**
- **SIGPIPE (13) Ecriture sur un conduit non-ouvert en lecture.**
- **SIGALRM (14) Horloge: émis quand l'horloge d'un processus s'arrête. L'horloge est mise en marche par la primitive `alarm()`.**
- **SIGTERM (15) Terminaison logicielle: émis lors de la terminaison normale d'un processus. Il est également utilisé lors d'un arrêt du système pour mettre fin à tous les processus actifs.**
- **SIGUSR1 (16) Premier signal à la disposition de l'utilisateur: utilisé pour la communication inter-processus.**
- **SIGUSR2 (17) Deuxième signal à la disposition de l'utilisateur: idem SIGUSR1.**

- **SIGCLD (18) Mort d'un fils: envoyé au père à la terminaison d'un processus fils.**

- **SIGPWR (19) Réactivation sur panne d'alimentation.**

Remarque:

Les signaux repérés par * générent un fichier `core` sur le disque, lorsqu'ils ne sont pas traités correctement.

Pour plus de portabilité, on peut écrire des programmes utilisant des signaux en appliquant les règles suivantes: éviter les signaux SIGIOT, SIGEMT, SIGBUS et SIGSEGV qui dépendent de l'implémentation. Il est correct de les intercepter pour imprimer un message, mais il ne faut pas essayer de leur attribuer une quelconque signification.

3.1.3 Signal SIGCLD: gestion des processus zombi

Le signal SIGCLD se comporte différemment des autres. S'il est ignoré, la terminaison d'un processus fils, alors que le processus père n'est pas en attente, n'entraînera pas la création de processus zombi (voir en 2.2.3).

Exemple:

Le programme suivant génère un zombi, lorsque le père reçoit à la mort de son fils un signal SIGCLD.

```
                /* fichier test_sigcld.c */

#include <stdio.h>

main()
{
     if (fork() != 0) {
          while(1) ;
     }
}
```

Résultat de l'exécution:

```
obernard/> test_sigcld &
obernard/> ps -a
   PID TTY      TIME COMMAND
  8507          0:00 <defunct>
```

Dans le programme qui suit, le père ignore le signal SIGCLD, et son fils ne devient plus un zombi.

```
                /* fichier test_sigcld2.c */

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

main()
{
     signal(SIGCLD,SIG_IGN) ;/* ignore le signal SIGCLD */
     if (fork() != 0) {
          while(1) ;
     }
}
```

Résultat de l'exécution:

```
obernard/> test_sigcld2 &
obernard/> ps -a
   PID TTY      TIME COMMAND
```

Remarque: la primitive `signal()` sera détaillée en 3.2.2.

3.1.4 Signal SIGHUP: gestion des applications longues

Ce signal peut être gênant lorsqu'on désire qu'un processus se poursuive après la fin de la session de travail (application longue). En effet, si le processus ne traite pas ce signal, il sera interrompu par le système au moment du délogage.
Différentes solutions se présentent:

1. **Utiliser la commande shell `at`, qui permet de lancer l'application à une certaine date, via un processus du système, appelé *démon*. Dans ce cas, le processus n'étant attaché à aucun terminal, le signal SIGHUP sera sans effet.**

2. **Inclure dans le code de l'application la réception du signal SIGHUP.**

3. **Lancer en arrière-plan (en effet un processus lancé en background traite automatiquement le signal SIGHUP).**

4. **Lancer l'application sous le contrôle de la commande `nohup`, qui entraîne un appel à `trap`, et redirige la sortie standard sur `nohup.out`.**

3.2 Traitement des signaux

3.2.1 Emission d'un signal

Primitive kill()

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

```
int kill(pid,sig)    /* emission d'un signal */
int pid ;            /* identificateur du processus ou du groupe destinataire */
int sig ;            /* numero du signal */
```

Valeur retournée: 0 si le signal a été envoyé, -1 sinon.

Cette primitive émet à destination du processus de numéro *pid* le signal de numéro *sig*.
De plus, si l'entier *sig* est nul, aucun signal n'est envoyé, et la valeur de retour permet de savoir si le nombre *pid* est un numéro de processus ou non.

Utilisation du paramètre *pid*:

- Si *pid* $>$ 0: *pid* désigne alors le processus d'identificateur *pid*.
- Si *pid* $=$ 0: le signal est envoyé à tous les processus du même groupe que l'émetteur (cette possibilité est souvent utilisée avec la commande shell `kill`
  `kill -9 0`
  pour tuer tous les processus en arrière-plan sans avoir à indiquer leurs identificateurs de processus).
- Si *pid* $=$ -1:
  - si le processus appartient au super-utilisateur, le signal est envoyé à tous les processus, sauf aux processus système et au processus qui envoie le signal.
  - sinon, le signal est envoyé à tous les processus dont l'identificateur d'utilisateur réel est égal à l'identificateur d'utilisateur effectif du processus qui envoie le signal (c'est un moyen de tuer tous les processus dont on est propriétaire, indépendamment du groupe de processus).
- Si *pid* $<$ -1: le signal est envoyé à tous les processus dont l'identificateur de groupe de processus (*pgid*) est égal à la valeur absolue de *pid*.

Notons que la primitive `kill()` est le plus souvent exécutée via la commande shell `kill`.

Primitive alarm()

```
#include <signal.h>

unsigned alarm(secs)          /* envoi d'un signal SIGALRM */
unsigned secs ;               /* nombre de secondes */
```

Valeur retournée: temps restant dans l'horloge.

Cette primitive envoie un signal SIGALRM au processus appelant après un laps de temps *secs* (en secondes) passé en argument, puis réinitialise l'horloge d'alarme. A l'appel de la primitive, l'horloge est initialisée à *secs* secondes, et est décrémentée jusqu'à 0.
Si le paramètre *secs* est nul, toute requête est annulée.
Cette primitive peut être utilisée, par exemple, pour forcer la lecture au clavier dans un délai donné.
Le traitement du signal doit être prévu, sinon le processus est tué.

Exemple 1:

```
                /* fichier test_alarm.c */

/*
 * test des valeurs de retour de alarm()
 * ainsi que de son fonctionnement
 */

#include <errno.h>
#include <signal.h>

it_horloge(sig) /* routine executee sur reception de SIGALRM */
int sig ;
{
     printf("reception du signal %d : SIGALRM\n",sig) ;
}

main()
{
unsigned sec ;
     signal(SIGALRM,it_horloge) ; /* interception du signal */
     printf("on fait alarm(5)\n") ;
     sec=alarm(5) ;
     printf("valeur retournee par alarm : %d\n",sec) ;
     printf("le principal boucle a l'infini (CTRL-c pour arreter)\n") ;
     for(;;) ;
}
```

Résultat de l'exécution:

```
on fait alarm(5)
valeur retournee par alarm : 0
le principal boucle a l'infini (CTRL-c pour arreter)
reception du signal 14 : SIGALRM
```

Exemple 2:

```
                /* fichier test_alarm2.c */

/*
 * test des valeurs de retour de alarm()
 * lorsqu'on fait 2 alarm() successifs
 */

#include <errno.h>
#include <signal.h>

void it_horloge(sig) /* traitement lors du deroutement sur SIGALRM */
int sig ;
{
     printf("reception du signal %d : SIGALRM\n",sig) ;
     printf("attention, le principal reprend la main\n") ;
}

void it_quit(sig) /* traitement lors du deroutement sur SIGINT */
int sig ;
{
     printf("reception du signal %d : SIGINT\n",sig) ;
     printf("pourquoi moi ?\n") ;
     exit(1) ;
}

void main()
{
int i ;
unsigned sec ;
     signal(SIGINT,it_quit) ; /* interception du ctrl-c */
     signal(SIGALRM,it_horloge) ; /* interception signal d'alarm */
     printf("on arme alarm pour dans 10 secondes\n") ;
     sec=alarm(10) ;
     printf("valeur retournee par alarm : %d\n",sec) ;
     printf("on fait patienter 3 secondes avec sleep\n") ;
     sleep(3) ;
     printf("on refait alarm(5) avant l'arrivee du signal precedent\n") ;
     sec=alarm(5) ;
     printf("nouvelle valeur retournee par alarm : %d\n",sec) ;
     printf("le principal boucle a l'infini (ctrl-c pour arreter)\n") ;
     for(;;) ;
}
```

Remarque:

L'interception du signal SIGINT n'a pour seul but ici que de fournir une manière élégante de sortir du programme et donc, de permettre une redirec-

tion de la sortie standard vers un fichier résultat.

Résultat de l'exécution:

```
on arme alarm pour dans 10 secondes
valeur retournee par alarm : 0
on fait patienter 3 secondes avec sleep
on refait alarm(5) avant l'arrivee du signal precedent
nouvelle valeur retournee par alarm : 7
le principal boucle a l'infini (ctrl-c pour arreter)
reception du signal 14 : SIGALRM
attention, le principal reprend la main
reception du signal 2 : SIGINT
pourquoi moi ?
```

On peut remarquer, ici, que l'horloge est réinitialisée à 5 secondes lors du deuxième appel à `alarm()`, et que, de plus, la valeur retournée est l'état de l'horloge.
Enfin, on peut observer que l'horloge se décremente au cours du temps.

Remarque:

La fonction `sleep()` fait appel à `alarm()`. Il faudra donc l'utiliser avec prudence si l'on manipule déjà **SIGALRM**.

Exemple d'implémentation de `sleep()`:

Le programme ci-dessous réalise l'implémentation d'une version de la fonction `sleep()` qui utilise les primitives `pause()` et `alarm()`.

Principe:

Un processus arme une alarme et se met en pause. A l'arrivée du signal **SIGALRM**, `pause()` est interrompue et le processus termine son exécution.

```
                 /* fichier test_sleep.c */

/*
 * utilisation de pause() et alarm() pour
 * realiser une fonction sleep2
 */

#include <erno.h>
#include <signal.h>

#define BADSIG (int (*)())-1 /* on definit la valeur de retour en cas d'erreur
                              * de la primitive signal(), soit :
                              * -1 force a un pointeur sur une fonction qui
```

```
                             * retourne un int
                             */

void nullfcn()  /* on definit quand meme la fonction executee
                 * lors de l'interception de SIGALRM par signal()
                 * cette fonction ne fait rien du tout
                 */
{
}

void sleep2(secs) /* endort pour secs secondes */
int secs ;
{
     if(signal(SIGALRM,nullfcn)==BADSIG)
     {
          perror("erreur reception signal") ;
          exit(-1) ;
     }
     alarm(secs) ; /* initialise l'horloge a secs secondes */
     pause() ;     /* on se met en attente d'un signal */
}

main() /* juste pour tester sleep2() */
{
     if(fork()==0) /* fils */
     {
          sleep(3) ;
          printf("hello, sleep\n") ;
     }
     else /* pere */
     {
          sleep2(3) ;
          printf("hello, sleep2\n") ;
     }
}
```

Résultat de l'exécution:

Au bout de 3 secondes on a indifféremment:

```
hello, sleep2
hello, sleep
```

ou bien:

```
hello, sleep
hello, sleep2
```

Remarque:

L'interêt de `nullfcn()` est de s'assurer que le signal de réveil ne provoque pas le comportement par défaut et n'est pas non plus ignoré, afin que `pause()` puisse être interrompue.

3.2.2 Réception des signaux

Primitive signal()

```
#include <signal.h>

int (*signal(sig,fcn)) ()     /* reception d'un signal */
int sig ;                     /* numero du signal */
int (*fcn)() ;                /* action apres reception */
```

Valeur retournée: adresse de la fonction spécifiant le comportement du processus vis-à-vis du signal considéré, -1 sinon.

Cette primitive intercepte le signal de numéro *sig*. Le second argument est un pointeur sur une fonction qui peut prendre une des trois valeurs suivantes:

1. **SIG_DFL: ceci choisit l'action par défaut pour le signal. La réception d'un signal par un processus entraîne alors la terminaison de ce processus, sauf pour SIGLD et SIGPWR, qui sont ignorés par défaut. Dans le cas de certains signaux, il y a création d'un fichier image `core` sur disque.**
2. **SIG_IGN: ceci indique que le signal doit être ignoré: le processus est immunisé. On rappelle que le signal SIGKILL ne peut être ignoré.**
3. **un pointeur sur une fonction (nom de la fonction): ceci implique le captage du signal. La fonction est appelée quand le signal arrive, et après son exécution, le traitement du processus reprend où il a été interrompu. On ne peut procéder à un déroutement sur la réception d'un signal SIGKILL puisque ce signal n'est pas interceptable.**

Nous voyons donc qu'il est possible de modifier le comportement d'un processus à l'arrivée d'un signal donné. C'est ce qui se passe pour un certain nombre de processus standards: le shell, par exemple, à la réception d'un signal SIGINT affiche le prompt '$' (et n'est pas interrompu).

Primitive pause()

```
void pause()             /* attente d'un signal quelconque */
```

Cette primitive correspond à de l'attente pure. Elle ne fait rien, et n'attend rien de particulier. Cependant, puisque l'arrivée d'un signal interrompt toute primitive bloquée, on peut tout aussi bien dire que `pause()` attend un signal. On observe alors le comportement de retour classique d'une primitive bloquée,

c'est à dire le positionnement de `errno` à EINTR.
Notons que le plus souvent, le signal que `pause()` attend est l'horloge d'`alarm()`.

Exemple:

```
                /* fichier test_pause.c */

/*
 * test sur la valeur retournee par pause()
 */

#include <errno.h>
#include <signal.h>

void it_main(sig) /* traitement sur 1er SIGINT */
int sig ;
{
     printf("reception du signal numero : %d\n",sig) ;
     printf("faut vraiment retourner en cours ?\n") ;
     printf("c'est la que les profs insistent generalement!\n") ;
}

void it_fin(sig) /* traitement sur 2eme SIGINT */
int sig ;
{
     printf("reception du signal numero : %d\n",sig) ;
     printf("bon ok,ca va, ca va ...\n") ;
     exit(1) ;
}

void main()
{
     signal(SIGINT,it_main) ; /* interception 1er SIGINT */
     printf("on va faire une petite pause (cafe-clop)\n") ;
     printf("tapez CTRL-c pour jouer au prof\n") ;
     printf("retour de pause (sur reception signal) : %d\n",pause()) ;
     printf("errno = %d\n",errno) ;
     signal(SIGINT,it_fin) ; /* on rearme l'interception : 2eme SIGINT */
     for(;;) ;
}
```

Résultat de l'exécution:

```
on va faire une petite pause (cafe-clop)
tapez CTRL-c pour jouer au prof
reception du signal numero : 2
faut vraiment retourner en cours ?
c'est la que les profs insistent generalement!
```

```
retour de pause (sur reception signal) : -1
errno = 4
reception du signal numero : 2
bon ok,ca va, ca va ...
```

La primitive `pause()` est interrompue par le signal SIGINT, retourne -1 et `errno` est positionné à 4 : `interrupted system call`.

3.3 Exemples

3.3.1 Héritage des signaux par fork()

Les processus fils recevant l'image mémoire du père, héritent de son comportement vis-à-vis des signaux. L'exemple suivant décrit ce phénomène:

```
                 /* fichier test_sign_fork.c */

/*
 * heritage par le fils du comportement du pere
 * vis-a-vis des signaux
 */

#include <errno.h>
#include <signal.h>

main()
{
 int fin() ;
     signal(SIGQUIT,SIG_IGN) ;
     signal(SIGINT,fin) ;
     if (fork()>0){
          printf("processus pere : %d\n",getpid()) ;
          while(1) ;
     }
     else {
          printf("processus fils : %d\n",getpid()) ;
          while(1) ;
     }
}
fin()
{
     printf("SIGINT pour le processus %d\n",getpid()) ;
     exit(0) ;
}
```

Résultat de l'exécution:

```
obernard/> test_sign_fork
```

```
processus fils : 9180
processus pere : 9179
^D   <-----pas de reaction
^C
SIGINT pour le processus 9180
SIGINT pour le processus 9179
```

3.3.2 Communication entre processus

Il s'agit d'un exemple simple utilisant les primitives d'émission et de réception de signaux, afin de faire communiquer deux processus entre eux. En outre, l'exécution de ce programme permet de s'assurer que le processus exécutant la routine de déroutement est bien celui qui re,coit le signal.

```
                /* fichier test_kill_signal.c */

/*
 * communication simple entre deux processus
 *  au moyen des signaux
 */

#include <errno.h>
#include <signal.h>

void it_fils()
{
     printf("- oui, et je le ferai moi meme... ARGHH...\n") ;
     kill (getpid(),SIGINT) ;
}

void fils()
{
     signal(SIGUSR1,it_fils) ;
     printf("- allo maman bobo, comment tu m'as fait j'suis pas beau!!!\n") ;
     while(1) ;
}

main()
{
 int ppid, pid ;

     if ((pid=fork())==0) fils() ;
     else{
          sleep(3) ;
          printf("- mon fils, veux-tu rejoindre le domaine des morts?\n") ;
          kill (pid,SIGUSR1) ;
     }
}
```

Résultat de l'exécution:

```
obernard/> test_kill_signal
- allo maman bobo, comment tu m'as fait j'suis pas beau!!!
- mon fils, veux-tu rejoindre le domaine des morts?
- oui, et je le ferai moi meme... ARGHH...
```

Histoire courte:

Un processus crée un fils qui ne semble pas heureux de vivre. Il lui envoie alors un signal SIGUSR1. A la réception de ce dernier, le fils désespéré décide de s'envoyer un signal SIGINT, pour se suicider.

3.3.3 Contrôle de l'avancement d'une application

Tous ceux qui ont lancé des programmes de simulation ou de calcul numérique particulièrement longs ont sûrement désiré connaître l'état d'avancement de l'application pendant son exécution. Ceci est parfaitement réalisable grâce à la commande shell `kill`, en envoyant au processus concerné un signal; le processus peut alors à la réception d'un tel signal, afficher les données désirées. Voici un exemple qui permet de résoudre le problème:

```
                 /* fichier surveillance.c */

#include <errno.h>
#include <signal.h>

/* les variables a editer doivent etre globales */
long somme ;

void it_surveillance()
{
 long t_date ;
     signal(SIGUSR1, it_surveillance) ;/* reactive SIGUSR1 */
     time(&t_date) ;
     printf("\n date du test : %d ",ctime(&t_date)) ;
     printf("valeur de la somme : %d \n",somme) ;
}

main()
{
     signal(SIGUSR1,it_surveillance) ;
     printf ("Envoyez le signal a %d \n",getpid()) ;
     while(1) somme++ ;
}
```

Exécution:

Si on lance le programme en tâche de fond, il suffira de taper sous le shell la commande :

```
kill -16 pid
```

pour obtenir l'affichage des variables de contrôle.

3.4 Conclusion

A l'exception de SIGCLD, les signaux qui arrivent ne sont pas mémorisés. Ils sont ignorés, ils mettent fin aux processus, ou bien ils sont interceptés. C'est la raison principale qui rend les signaux inadaptés pour la communication inter-processus: un message sous forme de signal peut être perdu s'il est reçu à un moment où ce type de signal est temporairement ignoré. Lorsqu'un signal a été capté par un processus, le processus réadopte son comportement par défaut vis-à-vis de ce signal. Donc, si l'on veut pouvoir capter un même signal plusieurs fois, il convient de redéfinir le comportement du processus par la primitive `signal()`. Généralement, on réarme l'interception du signal le plus tôt possible (première instruction effectuée dans la procédure de traitement du déroutement).

Un autre problème est que les signaux sont plutôt brutaux: à leur arrivée, ils interrompent le travail en cours. Par exemple, la réception d'un signal pendant que le processus est en attente d'un événement (ce qui peut arriver lors de l'utilisation des primitives `open()`, `read()`, `write()`, `msgrcv()`, `pause()`, `wait()` ...), lance l'exécution de la fonction de déroutement; à son retour, la primitive interrompue renvoie un message d'erreur sans s'être exécutée totalement (`errno` est positionné à EINTR).
Par exemple, lorsqu'un processus père qui intercepte les signaux d'interruption et d'abandon, est en cours d'attente de la terminaison d'un fils, il est possible qu'un signal d'interruption ou d'abandon éjecte le père hors du `wait()` avant que le fils n'ait terminé (d'où la création d'un `<defunct>`). Une méthode pour remédier à ce type de problème, est d'ignorer certains signaux avant l'appel de telles primitives (ce qui pose alors d'autres problèmes, puisque ces signaux ne seront pas traités).

Chapter 4

LES PIPES ou TUBES DE COMMUNICATION

4.1 Introduction

Les pipes (ou conduits, tubes) constituent un mécanisme fondamental de communication unidirectionnelle entre processus. Ce sont des files de caractères (FIFO: First In First Out): les informations y sont introduites à une extrêmité et en sont extraites à l'autre.
Les conduits sont implémentés comme les fichiers (ils possèdent un i-node), même s'ils n'ont pas de nom dans le système.
La technique du conduit est fréquemment mise en œuvre dans le shell pour rediriger la sortie standard d'une commande sur l'entrée d'une autre (symbole |).

4.2 Particularités des tubes

- Comme ils n'ont pas de nom, les tubes de communication sont temporaires, ils n'existent que le temps d'exécution du processus qui les crée.
- De plus, leur création doit se faire à partir d'une primitive spéciale: `pipe()`.
- Plusieurs processus peuvent écrire et lire sur un même tube, mais aucun mécanisme ne permet de différencier les informations à la sortie.
- La capacité est limitée (en général à 4096 octets). Si on continue à écrire dans le conduit alors qu'il est plein, on se place en situation de blocage (dead-lock).
- Les processus communiquant au travers de tubes doivent avoir un lien de parenté, et les tubes les reliant doivent avoir été ouverts avant la création des fils (voir le passage des descripteurs de fichiers ouverts à l'exécution de `fork()`, en **2.2.1**).
- Il est impossible de se déplacer à l'intérieur d'un tube.

- Afin de faire dialoguer deux processus par tube, il est nécessaire d'ouvrir deux conduits, et de les utiliser l'un dans le sens contraire de l'autre.

4.3 Création d'un conduit

Primitive pipe()

```
int pipe(p_desc)            /* cree un tube */
int p_desc[2] ;             /* descripteurs d'ecriture et de lecture */
```

Valeur retournée: 0 si la création s'est bien passée, et -1 sinon.

`p_desc[0]` contient le numéro du descripteur par lequel on peut lire dans le tube.
`p_desc[1]` contient le numéro du descripteur par lequel on peut écrire dans le tube.
Ainsi, l'écriture dans `p_desc[1]` introduit les données dans le conduit, la lecture dans `p_desc[0]` les en extrait.

4.4 Sécurités apportées par le système

Dans le cas où tous les descripteurs associés aux processus susceptibles de lire dans un conduit sont fermés, un processus qui tente d'y écrire reçoit le signal SIGPIPE, et donc est interrompu s'il ne traite pas ce signal.

Si un tube est vide, ou si tous les descripteurs susceptibles d'y écrire sont fermés, la primitive `read()` renvoie la valeur 0 (fin de fichier atteinte).

4.4.1 Exemple 1: émission du signal SIGPIPE

```
                 /* fichier test_pipe_sig.c */

/*
 * teste l'ecriture dans un conduit ferme en lecture
 */

#include <errno.h>
#include <signal.h>

void it_sigpipe()
{
     printf("Reception du signal SIGPIPE\n") ;
}
main()
{
     int p_desc[2] ;
     signal(SIGPIPE,it_sigpipe) ;
```

```
     pipe(p_desc) ;
     close(p_desc[0]) ;  /* fermeture du conduit en lecture */
     if (write(p_desc[1],"0",1) == -1)
          perror("Erreur write") ;
}
```

Résultat de l'exécution:

```
obernard/> test_pipe_sig
 Reception du signal SIGPIPE
 Erreur write: Broken pipe
```

Dans cet exemple on essaie d'écrire dans le tube alors qu'on vient de le fermer en lecture; le signal SIGPIPE est émis, et le programme est dérouté. Au retour, la primitive `write()` renvoie -1 et `perror` affiche le message d'erreur.

4.4.2 Exemple 2: lecture dans un tube fermé en écriture

```
                  /* fichier test_pipe_read.c */

/*
 * teste la lecture dans un tube ferme en ecriture
 */

#include <errno.h>
#include <signal.h>

main()
{
  int i, ret, p_desc[2] ;
  char c ;
     pipe(p_desc) ; /* creation du tube */
     write(p_desc[1],"AB",2) ; /* ecriture de deux lettres dans le tube */
     close(p_desc[1]) ;        /* fermeture du tube en ecriture */

     /* tentative de lecture dans le tube */
     for (i=1; i<=3,i++) {
          ret = read(p_desc[0],&c,1) ;
          if (ret = 1)
               printf("valeur lue: %c\n",c) ;
          else
               perror("impossible de lire dans le tube\n") ;
     }
}
```

Résultat de l'exécution:

```
obernard/> test_pipe_read
```

```
valeur lue:A
valeur lue:B
impossible de lire dans le tube: Not a typewriter
```

Cet exemple montre que la lecture dans le tube est possible même si celui-ci est fermé en écriture. Bien-sûr, lorsque le tube est vide, `read()` renvoie la valeur 0.

4.5 Application des primitives d'entrée/sortie aux tubes

Il est possible d'utiliser les fonctions de la bibliothèque standard sur un tube ayant été ouvert, en lui associant, au moyen de la fonction `fdopen()`, un pointeur sur un objet de structure FILE.

- `write()`: les données sont écrites dans le conduit dans leur ordre d'arrivée. Lorsque le conduit est plein, `write()` se bloque en attendant qu'une place se libère. On peut éviter ce blocage en positionnant le drapeau O_NDELAY.
- `read()`: Les données sont lues dans le conduit dans leur ordre d'arrivée. Une fois extraites, les données ne peuvent pas être relues ou restituées au conduit.
- `close()`: Cette fonction a plus d'importance sur un conduit que sur un fichier. Non seulement elle libère le descripteur de fichier, mais lorsque le descripteur de fichier d'écriture est fermé, elle agit comme une fin de fichier pour le lecteur.
- `dup()`: Cette primitive combinée avec la primitive `pipe()` permet d'implémenter des commandes reliées par des tubes, en redirigeant la sortie standard d'une commande vers l'entrée standard d'une autre.

4.6 Exemples globaux

4.6.1 Implémentation d'une commande pipée

Cette exemple permet d'observer comment combiner les primitives `pipe()` et `dup()` afin de réaliser des commandes shell du type `ls|wc|wc`.
Notons qu'il est nécessaire de fermer les descripteurs non utilisés par les processus exécutant la routine.

```
                /* fichier test_pipe.c */

/*
 * ce programme est equivalent a la commande shell ls|wc|wc
 */
```

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

int p_desc1[2] ;
int p_desc2[2] ;

void faire_ls()
{
     /* sortie standard redirigee vers le premier tube */
     close (1) ;
     dup(p_desc1[1]) ;
     close(p_desc1[1]) ;

     /* fermeture des descripteurs non utilises */
     close(p_desc1[0]) ;
     close(p_desc2[1]) ;
     close(p_desc2[0]) ;

     /* execute la commande */
     execlp("ls","ls",0) ;
     perror("impossible d'executer ls ") ;
}

void faire_wc1()
{
     /* redirection de l'entree standard vers le premier tube */
     close(0) ;
     dup(p_desc1[0]) ;
     close(p_desc1[0]) ;
     close(p_desc1[1]) ;

     /* redirection de la sortie standard vers le second tube */
     close(1) ;
     dup(p_desc2[1]) ;
     close(p_desc2[1]) ;
     close(p_desc2[0]) ;

     /* execute la commande */
     execlp("wc","wc",0) ;
     perror("impossible d'executer le premier wc") ;

}

void faire_wc2()
{
     /* redirection de l'entree standard vers le second tube */
     close (0) ;
```

```
     dup(p_desc2[0]) ;
     close(p_desc2[0]) ;

     /* fermeture des descripteurs non utilises */
     close(p_desc2[1]) ;
     close(p_desc1[1]) ;
     close(p_desc1[0]) ;

     /* execute la commande */
     execlp("wc","wc",0) ;
     perror("impossible d'executer le second wc") ;
}

main()
{
     /* creation du premier tube */
     if (pipe(p_desc1) == -1)
          perror("impossible de creer le premier tube") ;

     /* creation du second tube */
     if (pipe(p_desc2) == -1)
          perror("impossible de creer le second tube") ;

     /* lancement des fils */
     if (fork() == 0) faire_ls() ;
     if (fork() == 0) faire_wc1() ;
     if (fork() == 0) faire_wc2() ;
}
```

Résultat de l'exécution

```
obernard/> ls|wc|wc
      1       3     22
obernard/> test_pipe
      1       3     22
```

4.6.2 Communication entre père et fils grâce à un tube

Exemple 1: envoi d'un message à l'utilisateur

Ce programme permet à des processus d'envoyer des messages à l'aide de la messagerie électronique `mail`.

```
                 /* fichier test_pipe_mail.c */

/*
 * ce programme permet a un processus d'envoyer
 * un message a l'utilisateur
```

```
 */

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

main()
{
    FILE *fp;
    int pid, pipefds[2];
    char *username, *getlogin();

    /*
     * donne le nom d'utilisateur
     */
    if ((username = getlogin()) == NULL) {
        fprintf(stderr, "qui etes-vous?\n");
        exit(1);
    }

    /*
     * Cree un tube. Ceci doit etre fait avant le fork().
     */
    if (pipe(pipefds) < 0) {
        perror("Erreur pipe");
        exit(1);
    }

    if ((pid = fork()) < 0) {
        perror("Erreur fork");
        exit(1);
    }

    /*
     * Code du fils:
     * le fils execute la commande mail, et donc envoie au
     * username le message contenu dans le tube
     */
    if (pid == 0) {
        /*
         * redirige le standard input vers le tube; la commande
         * executee par la suite aura comme entree (un message)
         * la lecture du tube
         */
        close(0);
        dup(pipefds[0]);
        close(pipefds[0]);
```

```
        /*
         * ferme le cote ecriture du tube, pour pouvoir voir
         * la sortie sur l'ecran
         */
        close(pipefds[1]);
        /*
         * execute la commande mail
         */
        execl("/bin/mail", "mail", username, 0);
        perror("Erreur exec");
        exit(1);
    }

    /*
     * Code du pere
     * ecriture d'un message dans le tube
     */

    close(pipefds[0]);
    fp = fdopen(pipefds[1], "w");
    fprintf(fp, "Hello from your program.\n");
    fclose(fp);

    /*
     * Attente de la mort du processus fils
     */
    while (wait((int *) 0) != pid)
        ;

    exit(0);
}
```

L'utilisateur qui exécute ce programme se voit adresser un message.

Exemple 2: mise en évidence de l'héritage des descripteurs lors d'un fork()

Dans l'exemple suivant, un processus crée un conduit, crée un fils, écrit un texte dans le tube. Le fils hérite du conduit et de ses descripteurs; il effectue une lecture dans le tube. Mais attention, si le fils hérite des descripteurs, il n'en connait pas pour autant les numéros (le code du père et celui du fils étant dans deux procédures différentes). Il est donc indispensable de passer ces numéros (ou celui qui est utilisé) en paramètre.

```
                /* fichier test_pipe_fork.c */

/*
 * teste l'heritage des descripteurs lors de l'appel a fork()
```

```
 */

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

void code_fils(numero)
int numero;
{
 int fd ;
 int nread ;
 char texte[100] ;

     fd = numero ;
     printf("le descripteur est:%d\n",fd) ;
    /*
     * lecture dans le tube
     */
     switch (nread=read(fd,texte,sizeof(texte))) {
     case -1:
          perror("Erreur read") ;
     case 0:
          perror("Erreur EOF") ;
     default:
          printf("lit %d octets:%s\n",nread,texte) ;
     }
}

main()
{
 int fd[2] ;
 char chaine[10] ;
    /*
     * creation d'un tube
     */
     if (pipe(fd)==-1)
          perror("Erreur ouverture du tube") ;
    /*
     * creation d'un fils
     */
     switch (fork()){
     case -1:
          perror("Erreur fork") ;
     case 0:
          if (close(fd[1]) == -1) /* fermeture du descripteur */
                                  /*  non utilise */
               perror("Erreur close") ;
          code_fils(fd[0]) ;
```

```
          exit(0) ;
     }
    /*
     * ecriture dans le tube
     */
     close(fd[0]) ;
     if(write(fd[1],"hello",6)==-1)
          perror("Erreur write") ;
}
```

Resultat de l'exécution:

```
obernard/> test_pipe_fork
  le descripteur est: 3
 lit 6 octets : hello
```

Notons que comme le fils lit dans le conduit sans y écrire, il commence par fermer l'extrêmité d'écritrure (`fd[1]`) pour économiser les descripteurs de fichier. De même, comme le père ne se sert pas de l'extrêmité de lecture du conduit, il ferme le descripteur correspondant (`fd[0]`). Puis il écrit 6 octets dans le conduit.

4.7 Conclusion

Il est possible pour un processus d'utiliser lui-même un tube à la fois en lecture et en écriture; ce tube n'a plus alors son rôle de communication mais devient une implémentation de la structure de file. Cela permet, sur certaines machines, de dépasser la limite de taille de zone de données.
Le mécanisme de communication par tubes présente un certain nombre d'inconvénients comme la non-rémanance de l'information dans le système et la limitation de la classe de processus pouvant s'échanger des informations.

Chapter 5

LES FIFOs ou CONDUITS NOMMES

5.1 Introduction

Un FIFO combine les propriétés des fichiers et des conduits:

- comme un fichier, il a un nom, et tout processus ayant les autorisations appropriées peut l'ouvrir en lecture ou en écriture (même s'il n'a pas de lien de parenté avec le créateur du conduit). Ainsi, un tube nommé, s'il n'est pas détruit, persiste dans le système, même après la terminaison du processus qui l'a créé.
- une fois ouvert, un FIFO se comporte plutôt comme un conduit que comme un fichier: les données écrites sont lues dans l'ordre "First In First Out", sa taille est limitée, et il est impossible de se déplacer à l'intérieur.

5.2 Création d'un conduit nommé: primitive mknod()

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(path, perm)        /* creation d'un conduit nomme */
char *path ;                 /* nom du conduit */
int perm ;                   /* droits d'acces */
```

Valeur retournée: 0 si la création se déroule bien, -1 sinon.

La création de conduits nommés est le seul cas où un utilisateur normal a le droit d'utiliser cette primitive, réservée habituellement au super-utilisateur. Afin que l'appel de `mknod()` réussisse, il est indispensable que le drapeau S_IFIFO soit positionné dans le paramètre *perm* indiquant les droits d'accès: cela indique au système qu'un FIFO va être créé, et donc que l'utilisateur peut utiliser `mknod()`, même s'il n'est pas super-utilisateur.

A partir d'un programme, on peut éliminer un conduit nommé en utilisant la primitive `unlink(path)`, où `path` est le nom du conduit.

Exemple:

```
                /* fichier test_fifo.c */

#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

main()
{
     printf("Je vais creer un conduit de nom 'fifo1'\n") ;
     printf("Je vais creer un conduit de nom 'fifo2'\n") ;
     if (mknod("fifo1",S_IFIFO | 0666) == -1) {
          perror("impossible de creer le fifo") ;
          exit(1) ;
     }
     if (mknod("fifo2",S_IFIFO | 0666) == -1) {
          perror("impossible de creer le fifo") ;
          exit(1) ;
     }
     sleep(10) ;
     printf("Je vais effacer le conduit nomme fifo1\n") ;
     unlink("fifo1") ;
}
```

Résultat de l'exécution:

On lance le programme en background, et on vérifie l'existence du conduit nommé `fifo` puis son élimination par la commande shell `ls -l fifo`:

```
bolz> test_fifo &
[1]     379
bolz> Je vais creer un conduit de nom 'fifo1'
Je vais creer un conduit de nom 'fifo2'
ls -l fifo*
prw-rw-r--   1 bolz     speinf         0 Jun 22 11:12 fifo1
prw-rw-r--   1 bolz     speinf         0 Jun 22 11:12 fifo2
bolz> Je vais effacer le conduit nomme fifo1
ls -l fifo*
prw-rw-r--   1 bolz     speinf         0 Jun 22 11:12 fifo2
[1] +  Done                     test_fifo &
```

Remarques:

- Notons la présence du bit p qui indique que `fifo` est un tube nommé.
- On voit bien que même après la mort du processus qui l'a créé, le conduit nommé `fifo2` reste présent dans le système.
- L'élimination d'un tube nommé peut également se faire à partir du shell, en utilisant la commande `rm`.

5.3 Manipulation des FIFOs

Les instructions `read()` et `write()` sont bloquantes:

- tentative de lecture dans un FIFO vide: le processus attend un remplissage suffisant du tube.
- tentative d'écriture dans un FIFO plein: le processus attend que le FIFO soit suffisamment vidé.

Ici encore, le positionnement du drapeau O_NDELAY permet de passer outre le phénomène de blocage (les fonctions `read()` et `write()` retournent alors une valeur nulle).

Part III

COMMUNICATION INTER-PROCESSUS: LES IPC

Chapter 6

INTRODUCTION AUX IPC

6.1 Généralités

Les sémaphores, les segments de mémoire partagée, et les files de messages sont trois types de mécanismes avancés de communication interprocessus, regroupés sous la dénomination "System V IPC". On rencontre, parmi les primitives système qui permettent d'accéder à ces mécanismes, ainsi que dans les informations que le noyau maintient à leur sujet, de nombreuses similitudes.

On notera que:

Chapter 7

LES SEMAPHORES

7.1 Introduction

Les sémaphores sont des objets d'IPC utilisés pour synchroniser des processus entre eux. Ils constituent aussi une solution pour résoudre le problème d'exclusion mutuelle, et permettent en particulier de régler les conflits d'accès concurrents de processus distincts à une même ressource. L'implémentation qui en est faite ressemble à une synchronisation "sleep/wakeup".

7.2 Principe

Pour créer un sémaphore, un utilisateur doit lui associer une clé. Le système lui renvoie un identificateur de sémaphore auquel sont attachés n sémaphores (ensemble de sémaphores), numérotés de 0 à (n-1). Pour spécifier un sémaphore, l'utilisateur devra alors indiquer l'identificateur de sémaphore et le numéro de sémaphore.
A chaque sémaphore est associée une valeur, toujours positive, que l'utilisateur va pouvoir incrémenter ou décrémenter du nombre qu'il souhaite. Soit N la valeur initiale, et n le nombre d'incrémentation de l'utilisateur:

- si $n > 0$ alors l'utilisateur augmente la valeur du sémaphore de n et continue en séquence.
- si $n < 0$
  - si $N+n \geq 0$ alors l'utilisateur diminue la valeur du sémaphore de $|n|$ et continue en séquence.
  - si $N+n < 0$ alors le processus de l'utilisateur se bloque, en attendant que $N+n \geq 0$.
- si $n = 0$
  - si $N = 0$ alors l'utilisateur continue en séquence.
  - si $N \neq 0$ alors le processus de l'utilisateur se bloque en attendant que $N = 0$.

Nous verrons que le blocage des processus est paramétrable, c'est à dire que l'on peut spécifier au système de ne pas bloquer les processus mais de simplement renvoyer un code d'erreur et continuer en séquence.
D'autre part, à chaque identificateur de sémaphore sont associés des droits d'accès. Ces droits d'accès sont nécessaires pour effectuer des opérations sur les valeurs des sémaphores. Ces droits sont inopérants pour les deux manipulations suivantes:

- la destruction d'un identificateur de sémaphore
- la modification des droits d'accès.

Pour cela, il faudra être soit super-utilisateur, soit créateur, soit propriétaire du sémaphore.
Notons que le bon fonctionnement des mécanismes suppose que les opérations effectuées sur les sémaphores sont indivisibles (non interruptibles).

7.3 Structures utiles: semid_ds, sem, sembuf

Chaque ensemble de sémaphores du système est associé à plusieurs structures. La donnée de ces structures n'est pas superflue, car elle permet de comprendre ce que provoquent les primitives `semget()`, `semctl()`, `semop()`, au niveau du système. Ces structures sont contenues dans `<sys/sem.h>`:

```
struct semid_ds {    /* une par ensemble de semaphores dans le systeme */
            struct ipc_perm  sem_perm; /* operations permises */
            struct sem  *sem_base;     /* pointeur sur le premier */
                                       /* semaphore de l'ensemble */
            ushort  sem_nsems;         /* nombre de semaphores dans l'ensemble */
            time_t  sem_otime;         /* date de la derniere operation semop()*/
            time_t  sem_ctime;         /* date de la derniere modification */
};

struct sem {         /* une pour chaque semaphore dans le systeme */
            ushort  semval;      /* valeur du semaphore */
            sema_t  semnwait;    /* attente d'une valeur non nulle */
            sema_t  semzwait;    /* attente d'une valeur nulle */
            pid_t   sempid;      /* pid du dernier processus ayant effectue */
                                 /* une operation sur le semaphore */
            ushort  semncnt;     /* nombre de processus en attente que semval*/
                                 /* soit superieure a la valeur courante */
            ushort  semzcnt;     /* nombre de processus en attente que */
                                 /* semval devienne nulle */
};

struct sembuf {
            ushort  sem_num;     /* numero du semaphore */
            short   sem_op;      /* operation a realiser */
```

```
            short   sem_flg;     /* indicateur d'operation */
};
```

7.4 Primitive semget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key, nsems, semflg)
key_t key ;
int nsems, semflg ;
```

Valeur retournée: l'identificateur de sémaphore *semid*, ou -1 en cas d'erreur.

L'appel système `semget()` **est utilisé pour créer un nouvel ensemble de sémaphores, ou pour obtenir l'identificateur de sémaphore d'un ensemble existant. Le premier argument *key* est une clé indiquant le nom numérique de l'ensemble de sémaphores. Le second *nsems* indique le nombre de sémaphores de l'ensemble. Le dernier *semflg* est un drapeau spécifiant les droits d'accès sur l'ensemble de sémaphores.**

7.4.1 Les valeurs possibles pour *key*

- **IPC_PRIVATE (=0): l'ensemble n'a alors pas de clé d'accès, et seul le processus propriétaire, ou le créateur a accès à l'ensemble de sémaphores.**
- **la valeur désirée de la clé de l'ensemble de sémaphores.**

7.4.2 Les valeurs possibles pour *semflg*

Ce drapeau est en fait la combinaison de différentes constantes prédéfinies, permettant d'établir les droits d'accès, et les commandes de contrôle (la combinaison est effectuée de manière classique à l'aide de OU '|'). Notons la similitude existant entre les droits d'accès utilisés pour les sémaphores, et les droits d'accès aux fichiers UNIX: on retrouve la notion d'autorisation en lecture ou écriture aux attributs utilisateur/groupe/autres. Le nombre octal défini en 1.4.2 pourra être utilisé (en forçant à 0 les bits de droite d'exécution 3, 6 et 9).
Les constantes prédéfinies dans `<sys/sem.h>`**, et** `<sys/ipc.h>` **sont:**

```
#define   SEM_A       0200     /* permission de modification */
#define   SEM_R       0400     /* permission en lecture */
#define   IPC_CREAT 0001000    /* creation d'un ensemble de semaphores */
#define   IPC_EXCL  0002000    /* associe au bit IPC_CREAT provoque un */
                               /* echec si le fichier existe deja */
```

7.4.3 Comment créer un ensemble de sémaphores

Pour créer un ensemble de sémaphores, les points suivants doivent être respectés:

- *key* doit contenir la valeur identifiant l'ensemble (différent de IPC_PRIVATE=0).
- *semflg* doit contenir les droits d'accès désirés, et la constante IPC_CREAT.
- si l'on désire tester l'existence d'un ensemble correspondant à la clé *key* désirée, il faut rajouter à *semflg* la constante IPC_EXCL. L'appel à `semget()` échouera dans le cas d'existence d'un tel ensemble.

Notons que lors de la création d'un ensemble de sémaphores, un certain nombre de champs de l'objet de structure `semid_ds` sont initialisés (propriétaire, modes d'accès).

Exemple:

Ce programme crée un ensemble de quatre sémaphores associé à la clé 123.

```
                /* fichier test_semget.c */

/*
 * exemple d'utilisation de semget()
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define CLE 123

main()
{
  int semid ;  /* identificateur des semaphores */
  char *path = "nom_de_fichier_existant" ;

    /*
     * allocation de quatre semaphores
     */
     if (( semid = semget(ftok(path,(key_t)CLE), 4,
                     IPC_CREAT|IPC_EXCL|SEM_R|SEM_A)) == -1) {
          perror("Echec de semget") ;
          exit(1) ;
     }
     printf(" le semid de l'ensemble de semaphore est : %d\n",semid) ;
     printf(" cet ensemble est identifie par la cle unique : %d\n"
```

```
                            ,ftok(path,(key_t)CLE)) ;
}
```

Résultat de l'exécution

```
obernard/> test_semget
 le semid de l'ensemble de semaphore est : 2
 cet ensemble est identifie par la cle unique : 2073103658
obernard/> ipcs
IPC status from /dev/kmem as of Fri Jun 21 11:08:06 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
s      2 0x7b910d2a --ra-------     obernard speinf
obernard/> test_semget
Echec de semget: File exists
```

7.5 Primitive semctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(semid, semnum, cmd, arg) ;
int semid, semnum, cmd ;
union semun {
              int val ;
              struct semid_ds *buf ;
              ushort array[] ;  /* tableau de taille egale au nombre */
                                /* de semaphores de l'ensemble */
} arg ;
```

Valeur retournée: elle dépend de la valeur de *cmd*:

- si *cmd* = **GETVAL:** valeur de `semval`
- si *cmd* = **GETPID:** valeur de `sem_pid`
- si *cmd* = **GETNCNT:** valeur de `sem_ncnt`
- si *cmd* = **GETZCNT:** valeur de `sem_zcnt`

Pour les autres valeurs de *cmd*, la valeur retournée est 0 en cas de réussite, et -1 en cas d'erreur.

L'appel système `semctl()` est utilisé pour examiner et changer les valeurs de sémaphores d'un ensemble de sémaphores. Elle a besoin de quatre arguments: un identificateur de l'ensemble de sémaphores (*semid*), le numéro du sémaphore à examiner ou à changer (*semnum*), un paramètre de commande (*cmd*), et une variable de type union (*arg*).

Les différentes commandes possibles

Les diverses commandes possibles pour `semctl()` sont décrites dans les fichiers `<sys/ipc.h>` et `<sys/sem.h>`.
Dans le premier fichier, on retrouve les commandes suivantes:

- IPC_RMID (0): L'ensemble de sémaphores identifié par *semid* est détruit. Seul le super-utilisateur ou un processus ayant pour numéro d'utilisateur `sem_perm.uid` peut détruire un ensemble. Tous les processus en attente sur les sémaphores détruits sont débloqués et renvoient un code d'erreur.
- IPC_SET (1): Donne à l'identificateur de groupe, à l'identificateur d'utilisateur, et aux droits d'accès de l'ensemble de sémaphores, les valeurs contenues dans le champ `sem_perm` de la structure pointée par *arg.buf*. On met également à jour l'heure de modification.
- IPC_STAT (2): La structure associée à `semid` est rangée à l'adresse pointée par *arg.buf*.

Dans le second fichier, on trouve les commandes:

- GETNCNT (3): La fonction retourne la valeur de `semncnt` qui est le nombre de processus en attente d'une incrémentation de la valeur d'un sémaphore particulier.
- GETPID (4): La fonction retourne le *pid* du processus qui a effectué la dernière opération sur un sémaphore particulier.
- GETVAL (5): La fonction retourne la valeur `semval` du sémaphore de numéro `semnum`.
- GETALL (6): Les valeurs `semval` de tous les sémaphores sont rangées dans le tableau dont l'adresse est dans *arg.array*.
- GETZCNT (7): La fonction retourne la valeur `semzcnt` qui est le nombre de processus en attente d'un passage à zéro de la valeur d'un sémaphore particulier.
- SETVAL (8): Cette action est l'initialisation de la valeur du sémaphore. La valeur `semval` du sémaphore de numéro `semnum` est mise à *arg.val*.
- SETALL (9): Les valeurs `semval` des `semnum` premiers sémaphores sont modifiées en concordance avec les valeurs correspondantes du tableau dont l'adresse est dans *arg.array*.

Exemple:

Ce programme suppose que le sémaphore appelé est déjà créé (en lançant auparavant l'exemple relatif à `semget()`).

```
                /* fichier test_semctl.c */

/*
 * exemple d'utilisation de semctl()
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define CLE 123
union semun {
              int val ;
              struct semid_ds *buf ;
              ushort array[4] ;
             } arg ;

main()
{
  int semid, val_sem ,val_pid;
  char *path = "nom_de_fichier_existant" ;

    /*
     * recuperation de l'identificateur
     * de l'ensemble de semaphores de cle 123
     */
     if (( semid = semget(ftok(path,(key_t)CLE),0,0)) == -1 ) {
          perror ("Semget()") ;
          exit(1) ;
      }
      printf("L'ensemble de semaphore a comme semid : %d\n",semid) ;
      printf("La cle d'acces est %d\n",ftok(path,(key_t)CLE)) ;

    /*
     * mise a 1 du
     * troisieme semaphore
     */
     arg.val = 1 ;
     if ( semctl(semid,2,SETVAL,arg) == -1){
          perror("Semctl()") ;
          exit(1) ;
     }

    /*
     * lecture du
     * troisieme semaphore
```

```
     */
      if ( (val_sem = semctl(semid,2,GETVAL,arg)) == -1){
          perror("Semctl()") ;
          exit(1) ;
     }
     else printf("la valeur du troisieme semaphore est : %d\n",val_sem) ;

    /*
     * lecture du pid du processus qui a
     * effectue la derniere operation
     */
      if (( val_pid = semctl(semid,2,GETPID,arg) )== -1){
          perror("Semctl()") ;
          exit(1) ;
     }
     else printf("la valeur du pid du processus qui a effectue la derniere
                   operation est : %d,\nmon pid est :%d\n",val_pid,getpid()) ;

    /*
     * destruction du semaphore
     */
     if (semctl(semid,0,IPC_RMID,0)==-1){
             perror("impossible de detruire le semaphore") ;
             exit(1) ;
     }
}
```

Résultat de l'exécution:

```
obernard/> ipcs
IPC status from /dev/kmem as of Fri Jun 21 14:06:41 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
s    165 0x7b910d2a --ra-------     obernard   speinf
obernard/> test_semctl
L'ensemble de semaphore a comme semid : 165
La cle d'acces est 2073103658
la valeur du troisieme semaphore est : 1
la valeur du pid du processus qui a effectue la derniere operation est : 17392,
mon pid est :17392
obernard/> ipcs
IPC status from /dev/kmem as of Fri Jun 21 14:07:27 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
```

```
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
```

7.6 Primitive semop()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(semid, sops, nsops) ;
struct sembuf  (*sops)[] ;
int semid, nsops ;
```

Valeur retournée: la valeur `semval` du dernier sémaphore manipulé, ou -1 en cas d'erreur.

L'appel système `semop()` permet d'effectuer des opérations sur les sémaphores. Il utilise trois arguments: un identificateur d'ensemble de sémaphores (*semid*), un pointeur vers un tableau de structures de type `struct sembuf` (*sops*), et un entier donnant le nombre d'éléments de ce tableau (*nsops)*.
La structure `sembuf` spécifie le numéro du sémaphore qui sera traité, l'opération qui sera réalisée sur ce sémaphore, et les drapeaux de contrôle de l'opération.

Le type d'opération dépend de la valeur de *sem_op*:

- Si $sem\_op < 0$ (demande de ressource)
  - si $\texttt{semval} \geq |sem\_op|$ alors $\texttt{semval} = \texttt{semval} - |sem\_op|$: décrémentation du sémaphore
  - si $\texttt{semval} < |sem\_op|$ alors le processus se bloque jusqu'à ce que $\texttt{semval} \geq |sem\_op|$
- Si $sem\_op = 0$
  - si $\texttt{semval} = 0$ alors l'appel retourne
  - si $\texttt{semval} \neq 0$ alors le processus se bloque jusqu'à ce que $\texttt{semval} = 0$
- Si $sem\_op > 0$ (restitution de ressource): alors $\texttt{semval} = \texttt{semval} + sem\_op$

En effectuant des opérations `semop()` qui n'utilisent que des valeurs de *sem_op* égales à 1 ou -1, on retrouve le fonctionnement des sémaphores de DIJSKTRA (voir en 7.7).

System V fournit cependant une gamme d'opérations plus étendue en jouant sur la valeur de *sem_op*. L'implémentation est alors beaucoup plus complexe et la démonstration des garanties d'exclusion mutuelle est extrêmement délicate.

Le positionnement de certains drapeaux entraîne une modification du résultat des opérations de type `semop()`:

- **IPC_NOWAIT: évite le blocage du processus et renvoie un code d'erreur.**
- **SEM_UNDO: les demandes et restitutions de ressource sont automatiquement équilibrées à la fin du processus. Toutes les modifications faites sur les sémaphores par un processus sont défaites à la mort de celui-ci. Pendant toute la vie d'un processus, les opérations effectuées avec le drapeau SEM_UNDO sur tous les sémaphores de tous les identificateurs sont cumulées, et lors de la mort de ce processus, le système refait ces opérations à l'envers. Cela permet de ne pas bloquer indéfiniment des processus sur des sémaphores, après la mort accidentelle d'un processus. Notons que ce procédé coûte cher, tant au point de vue temps CPU qu'au point de vue réservation de place mémoire.**

Exemple:

Un premier processus (exécutant le programme `processus1`) crée un ensemble de sémaphores, fixe la valeur de l'un des sémaphores à 1, puis demande une ressource. Il se met en attente pendant 10 secondes. Un second processus (exécutant le programme `process2`) recupère l'identificateur *semid* de l'ensemble de sémaphores, puis demande également une ressource. Il reste bloqué jusqu'à ce que le premier processus ait fini son attente et libère alors la ressource.

```
                 /* fichier processus1.c */

/*
 * programme execute par le premier processus
 */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define CLE 123

int semid ;
struct sembuf operation[1] ;
char *path = "nom_de_fichier_existant" ;

union {
     int val ;
     struct semid_ds *buf ;
     ushort array[4] ; } arg ;

main()
{
```

```
/*
 * creation d'un ensemble de 4 semaphores
 */
 if (( semid = semget(ftok(path,(key_t)CLE),4,IPC_CREAT|SEM_A|SEM_R))==-1){
      perror("impossible de creer l'ensemble de semaphores") ;
      exit(1) ;
 }
 printf("process1: je viens de creer un ensemble de semphore : %d\n",semid) ;

/*
 * mise a 1 du troisieme semaphore
 */
 arg.val=1 ;
 if ((semctl(semid,2,SETVAL,arg))==-1){
     perror("semctl") ;
     exit(1);
     }

/*
 * demande de ressource au troisieme semaphore
 */
 printf("process1: je vais demander une ressource\n") ;
 operation[0].sem_num = 2 ; /* operation sur le troisieme semaphore */
 operation[0].sem_op = -1 ; /* operation de decrementation */
 operation[0].sem_flg = SEM_UNDO; /* pour defaire les operations */
 if ( semop(semid,operation,1) == -1){
      perror("semop:operation de decrementation non effectuee") ;
      exit(1) ;
 }

/*
 * attente pour bloquer le second processus
 */
 printf("process1: j'attends 10 sec\n") ;
 sleep(10) ; /* attente ... */
 printf("process1: j'ai fini d'attendre: je libere la ressource\n") ;

/*
 * liberation de ressource
 */
 operation[0].sem_op = 1 ; /* incrementation */
 if ( semop(semid,operation,1) == -1){
      perror("semop:operation d'incrementation non effectuee") ;
      exit(1) ;
 }
```

```
     printf("mort de process1\n") ;
     exit(0) ;
}


                 /* fichier processus2.c */

/*
 * programme execute par le second processus
 */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define CLE 123

int semid ;
struct sembuf operation[1] ;
char *path = "nom_de_fichier_existant" ;

main()
{

    /*
     * recuperation du semid
     */
     if (( semid = semget(ftok(path,(key_t)CLE),0,0))==-1){
          perror("impossible de retrouver l'ensemble de semaphores") ;
          exit(1) ;
     }
     printf("process2: traite les sem : semid %d\n",semid) ;


     /*
     * boucle d'attente de la disponibilite du semaphore.
     * On demande de ne pas rester bloquer en attente
     * en positionnement le drapeau IPC_NOWAIT
     */
     operation[0].sem_num = 2 ;
     operation[0].sem_op = -1 ;
     operation[0].sem_flg = IPC_NOWAIT + SEM_UNDO ;
     for (;;){
          if ( semop(semid,operation,1) != -1) break ;
          printf(" demande du process2 : semaphore non disponible\n") ;
```

```
          sleep(1) ;
     }
     printf(" semaphore alloue au process2\n") ;

    /*
     * liberation du segment de semaphore
     */
     if (semctl(semid,0,IPC_RMID,0) == -1){
          perror("probleme lors de la destruction des semaphores") ;
          exit(1) ;
     }
}
```

Résultat de l'exécution

Les deux programmes `processus1` et `processus2` sont lancés en background.

```
obernard/> processus1 &
[1]     20830
process1: je viens de creer un ensemble de semaphores : 385
process1: je vais demander une ressource
process1: j'attends 10 sec
obernard/> processus2 &
[2]     20845
process2: traite les sem : semid 385
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
demande du process2 : semaphore non disponible
process1: j'ai fini d'attendre: je libere la ressource
mort de process1
 semaphore alloue au process2
[2] + Done     process1&
[1] + Done     process2&
```

7.7 Sémaphores de Dijsktra

7.7.1 Principe

Les sémaphores de Dijkstra sont une solution simple au problème de l'exclusion mutuelle.
On accède à ces sémaphores par les deux opérations P (acquisition) et V (libération).
Lorsqu'on réalise l'opération P sur un sémaphore, sa valeur s est décrémentée

de 1 si s est différent de 0; sinon, le processus appelant est bloqué et est placé dans une file d'attente liée au sémaphore.
Lorsqu'on réalise l'opération V sur un sémaphore, on incrémente sa valeur s de 1 s'il n'y a pas de processus dans la file d'attente; sinon, s reste inchangée, et on libère le premier processus de la file.

7.7.2 Implémentation des sémaphores de Dijkstra

Le code ci-dessous réalise l'implémentation des sémaphores de Dijkstra à partir des mécanismes de sémaphores de System V.
La fonction `sem_create()` permet de créer un sémaphore. Les opérations P et V sont réalisées par les fonctions `P()` et `V()`. La fonction `sem_delete()` permet de détruire un sémaphore.

```
                 /* fichier dijkstra.h */

/*      Implementation des semaphores de Dijkstra a l'aide des semaphores
 *      de SystemV.
 *
 *      sem_create(): creation d'un semaphore
 *      P()         : realisation de l'operation P sur un semaphore
 *      V()         : realisation de l'operation V sur un semaphore
 *      sem_delete() : suppression d'un semaphore
 */

int sem_create(cle, initval)  /* creation d'un semaphore relie a cle */
                              /* de valeur initiale initval          */

key_t cle ;
int initval ;
{
     int semid ;
     union semun {
          int val ;
          struct semid_ds *buf ;
          ushort *array ;
     } arg_ctl ;

     semid = semget(ftok("dijkstra.h",cle),1,IPC_CREAT|IPC_EXCL|0666) ;
     if (semid == -1) {
          semid = semget(ftok("dijkstra.h",cle),1, 0666) ;
          if (semid == -1) {
               perror("Erreur semget()") ;
               exit(1) ;
          }
     }
```

```
     arg_ctl.val = initval ;
     if (semctl(semid,0,SETVAL,arg_ctl) == -1) {
          perror("Erreur initialisation semaphore") ;
          exit(1) ;
     }

     return(semid) ;
}

void P(semid)

int semid ;
{
     struct sembuf sempar ;

     sempar.sem_num = 0 ;
     sempar.sem_op = -1 ;
     sempar.sem_flg = SEM_UNDO ;
     if (semop(semid, &sempar, 1) == -1)
          perror("Erreur operation P") ;
}

void V(semid)

int semid ;
{
     struct sembuf sempar ;

     sempar.sem_num = 0 ;
     sempar.sem_op =  1 ;
     sempar.sem_flg = SEM_UNDO ;
     if (semop(semid, &sempar, 1) == -1)
          perror("Erreur operation V") ;
}

void sem_delete(semid)

int semid ;
{
     if (semctl(semid,0,IPC_RMID,0) == -1)
          perror("Erreur dans destruction semaphore") ;
}
```

7.7.3 Exemple d'utilisation des sémaphores de Dijkstra

L'exemple qui suit montre une utilisation simple de l'implémentation des sémaphores de Dijkstra réalisée ci-dessus.

Un processus est bloqué sur un sémaphore, et se débloque une fois que son fils libère la ressource.

```
                /*fichier test_sem_dijkstra.c */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include "dijkstra.h"

#define CLE 1

main()
{
  int sem ;

     sem = sem_create(CLE,0) ;
     printf("Creation du semaphore d'identificateur %d\n",sem) ;
     if (fork() == 0) {
          printf("Je suis le fils et j'attends 15 secondes...\n") ;
          sleep(15) ;
          printf("Je suis le fils et je fais V sur le semaphore\n") ;
          V(sem) ;
          exit(0) ;
     }
     else {
          printf("Je suis le pere et je me bloque en faisant P
                          sur le semaphore\n\n") ;
          P(sem) ;
          printf("Je suis le pere et je suis libre\n\n") ;
          sem_delete(sem) ;
     }
}
```

Résultat de l'exécution:

Le programme `test_sem_dijkstra` **est lancé en background, ce qui permet de vérifier l'existence dans le système du sémaphore créé, grâce à la commande shell** `ipcs`**:**

```
bolz> test_sem_dijkstra &
Creation du semaphore d'identificateur 275
[1]     1990
bolz> Je suis le fils et j'attends 15 secondes...
Je suis le pere et je me bloque en faisant P sur le semaphore

ipcs
IPC status from /dev/kmem as of Sat Jun 22 14:56:41 1991
```

```
T      ID      KEY         MODE        OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
s    275 0x01910d23 --ra-ra-ra-     bolz   speinf
bolz> Je suis le fils et je fais V sur le semaphore
Je suis le pere et je suis libre

ipcs
IPC status from /dev/kmem as of Sat Jun 22 14:56:57 1991
T      ID      KEY         MODE        OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
Semaphores:
[1] +  Done                    test_sem_dijkstra &
```

7.8 Conclusion

Le mécanisme des sémaphores sous System V est complexe à mettre en œ uvre. D'autre part, dans un programme utilisant des sémaphores, il faut être capable de démontrer que l'accès aux ressources partagées est exclusif, et qu'il n' y a ni interblocage, ni situation de famine (dans laquelle on n'obtient jamais d'accès). Si cette analyse est assez difficile avec les sémaphores de Dijkstra, elle devient extrêmement délicate avec les primitives de System V.

Chapter 8

LA MEMOIRE PARTAGEE

8.1 Introduction

Le partage de mémoire entre deux ou plusieurs processus (exécutant des programmes) constitue le moyen le plus rapide d'échange de données. La zone de mémoire partagée (appelée segment de mémoire partagée) est utilisée par chacun des processus comme si elle faisait partie de chaque programme. Le partage permet aux processus d'accéder à un espace d'adressage commun en mémoire virtuelle. De ce fait, il est dépendant du matériel de gestion mémoire, ce qui signifie que les fonctionnalités de ce type de communication inter-processus sont fortement liées au type de machine sur laquelle l'implémentation est réalisée.

8.2 Principe de la mémoire partagée

Un processus commence par créer un segment de mémoire partagée et ses structures de contrôle au moyen de la primitive `shmget()`. Lors de cette création, ce processus définit les droits d'opérations globaux pour le segment de mémoire partagée, définit sa taille en octets, et a la possibilité de spécifier que l'attachement d'un processus à ce segment de mémoire se fera en lecture seule. Pour pouvoir lire et écrire dans cette zone, il est nécessaire de posséder un identificateur de mémoire commune, noté *shmid*. Cet identificateur est fourni par le système, lors de l'appel à la primitive `shmget()` par tout processus donnant la clé associée.
Lorsqu'un segment de mémoire partagée est créé, un processus peut réaliser deux opérations:

- attachement de mémoire partagée, grâce à la primitive `shmat()`.
- détachement de mémoire partagée, grâce à la primitive `shmdt()`.

L'attachement de mémoire partagée permet au processus de s'associer avec le segment de mémoire partagée s'il en a le droit. Il récupère, en exécutant `shmat()`, un pointeur sur le début de la zone de mémoire qu'il peut utiliser

comme tout autre pointeur pour les lectures et les écritures.

Le détachement de mémoire partagée permet aux processus de se dissocier d'un segment de mémoire partagée lorsqu'il ne souhaite plus l'utiliser. A la suite de cette opération, ils perdent la faculté de lire ou d'écrire dans ce segment de mémoire partagée.

8.3 Structure associée à une mémoire commune: shmid_ds

```
struct shmid_ds
{ /* une pour chaque segment de memoire partagee dans le systeme */

     struct ipc_perm  shm_perm;     /* operations permises */
     int              shm_segsz;    /* taille du segment en octets */
     struct region   *shm_reg;      /* pointeur sur une region */
     ushort           shm_lpid;     /* pid du dernier processus ayant */
                                    /* effectue une operation */
     ushort           shm_cpid;     /* pid du processus createur */
     ushort           shm_nattch;   /* numero courant d'attachement */
     ushort           shm_cnattch;  /* numero attache en memoire */
     time_t           shm_atime;    /* date du dernier attachement */
     time_t           shm_dtime;    /* date du denrier detachement */
     time_t           shm_ctime;    /* date de la derniere modification */
} ;
```

8.4 Primitive shmget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key, size, shmflg)
key_t key ;
int size, shmflg ;
```

Valeur retournée: l'identificateur de mémoire partagée *shmid*, ou -1 en cas d'erreur.

Cette primitive est chargée de rechercher l'élément spécifié dans la structure `shmid_ds` **et, s'il n'existe pas, de le créer et de créer le morceau de mémoire physique demandé, mais sans générer d'adresse virtuelle pour lui.**
Elle prend trois arguments. Le premier (*key*) est une clé indiquant le nom numérique du segment de mémoire partagée. Le second (*size*) indique la taille (en octets) désirée pour le segment. Le dernier (*shmflg*) est un drapeau spécifiant les droits d'accès sur le segment.

8.4.1 Les valeurs possibles pour *key*

- **IPC_PRIVATE (=0): la zone mémoire n'a alors pas de clé d'accès, et seul le processus propriétaire, ou le créateur a accès au segment de mémoire partagée.**
- **la valeur désirée de la clé de la zone mémoire.**

8.4.2 Les valeurs possibles pour *shmflg*

On remarque que *shmflg* est semblable à *semflg*, utilisé pour les sémaphores. Ce drapeau est la combinaison de différentes constantes prédéfinies, permettant d'établir les droits d'accès, et les commandes de contrôle (la combinaison est effectuée de manière classique à l'aide de OU '|'). On retrouve la similitude existant entre les droits d'accès utilisés pour la mémoire partagée, et les droits d'accès aux fichiers UNIX: notion d'autorisation en lecture ou écriture aux attributs utilisateur/groupe/autres. Le nombre octal défini en 1.4.2 pourra être utilisé (en forçant à 0 les bits de droits d'exécution 3, 6 et 9).
Les constantes prédéfinies dans `<sys/shm.h>`, et `<sys/ipc.h>` sont:

```
#define   SHM_R          400   /* permission en lecture pour l'utilisateur */
#define   SHM_W          200   /* permission en lecture pour l'utilisateur */
#define   IPC_CREAT 0001000   /* creation d'un segment de memoire partagee */
#define   IPC_EXCL  0002000   /* associe au bit IPC_CREAT provoque */
                              /* un echec si le fichier existe deja */
```

8.4.3 Comment créer un segment de mémoire partagée

La procédure est identique à celle employée pour générer un ensemble de sémaphores: les points suivants doivent être respectés :

- *key* doit contenir la valeur identifiant le segment (différent de IPC_PRIVATE=0).
- *shmflg* doit contenir les droits d'accès désirés, et la constante IPC_CREAT.
- si l'on désire tester l'existence d'un segment correspondant à la clé désirée, il faut rajouter à *shmflg* la constante IPC_EXCL. L'appel `shmget()` échouera dans le cas d'existence d'un tel segment.

Notons que lors de la création d'un segment de mémoire partagée, un certain nombre de champs de l'objet de structure `shmid_ds` sont initialisés (propriétaire, modes d'accès).

Exemple d'utilisation de shmget():

Ce programme crée un segment de mémoire partagée associé à la clé 123, et vérifie le contenu des structures du système propres au segment.

```
                 /* fichier test_shmget.c */

/*
 * exemple d'utilisation de shmget()
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 123

main()
{
  int shmid ;  /* identificateur de la memoire commune */
  int size = 1000 ;
  char *path="test_shmget.c" ;

     if (( shmid = shmget(ftok(path,(key_t)CLE), size,
                    IPC_CREAT|IPC_EXCL|SHM_R|SHM_W)) == -1) {
          perror("Echec de shmget") ;
          exit(1) ;
     }
     printf("identificateur du segment: %d \n",shmid) ;
     printf("ce segment est associe a la cle unique: %d\n",
                   ftok(path,(key_t)CLE)) ;
}
```

Résultat de l'exécution:

On lance deux fois de suite le programme `test_shmget`**:**

```
bolz> test_shmget
identificateur du segment: 501
ce segment est associe a la cle unique: 2073103672
bolz> ipcs
IPC status from /dev/kmem as of Mon Jun 24 11:20:23 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m    100 0x0000004f --rw-rw-rw-     root     root
m    501 0x7b910d38 -Crw-------     bolz   speinf
Semaphores:
bolz> test_shmget
Echec de shmget: File exists
```

8.5 Primitive shmctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl(shmid, cmd, buf) ;
int shmid, cmd ;
struct shmid_ds *buf ;
```

Valeur retournée: 0 en cas de réussite, -1 sinon.

La primitive `shmctl()` est utilisée pour examiner et modifier des informations dans le segment de mémoire partagée. Elle prend trois arguments: un identificateur du segment de mémoire partagée (*shmid*), un paramètre de commande (*cmd*), et un pointeur vers une structure de type `shmid_ds` (*buf*).

Les différentes commandes possibles

On retrouve les commandes possibles pour `shmctl()` dans `<sys/ipc.h>` et `<sys/shm.h>`:

- **IPC_RMID (0): Le segment de mémoire identifié par *shmid* est détruit. Seul le super-utilisateur ou un processus ayant pour numéro d'utilisateur `shm_perm.uid` peut détruire un segment. Toutes les opérations en cours sur ce segment échoueront.**
- **IPC_SET (1): Donne à l'identificateur de groupe, à l'identificateur d'utilisateur, et aux droits d'accès du segment de mémoire, les valeurs contenues dans le champ `shm_perm` de la structure pointée par *buf*. On met également à jour l'heure de modification.**
- **IPC_STAT (2): La structure associée à *shmid* est rangée à l'adresse pointée par *buf*.**
- **SHM_LOCK (3): Verrouille en mémoire centrale le segment de mémoire partagée spécifié. Ce drapeau n'est utilisable que par le super-utilisateur. Ineffectif sur DPX2000.**
- **SHM_UNLOCK (4): Déverrouille de la mémoire centrale le segment de mémoire partagée. Ce drapeau n'est utilisable que par le super-utilisateur. Ineffectif sur DPX2000.**

Exemple:

Dans cet exemple, on suppose que le segment de mémoire partagée de clé 123 a été préalablement créé.

```
                 /* fichier test_shmctl.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 123

struct shmid_ds buf ;

main()
{
  char *path = "nom_de_fichier_existant" ;
  int shmid ;
  int size = 1000 ;

     /* recuperation de l'identificateur du segment de memoire de cle 123 */
     if (( shmid = shmget(ftok(path,(key_t)CLE),size,0)) == -1 ) {
          perror ("Erreur shmget()") ;
          exit(1) ;
     }

     /* lecture du pid du processus qui a effectue la derniere operation*/
     if ( shmctl(shmid,IPC_STAT,&buf) == -1){
          perror("Erreur shmctl()") ;
          exit(1) ;
     }
     printf("ETAT DU SEGMENT DE MEMOIRE PARTAGEE %d\n",shmid) ;
     printf("identificateur de l'utilisateur proprietaire: %d\n",buf.shm_perm.uid) ;
     printf("identificateur du groupe proprietaire: %d\n",buf.shm_perm.gid) ;
     printf("identificateur de l'utilisateur createur: %d\n",buf.shm_perm.cuid) ;
     printf("identificateur du groupe createur: %d\n",buf.shm_perm.cgid) ;
     printf("mode d'acces: %d\n",buf.shm_perm.mode) ;
     printf("taille de la zone memoire: %d\n",buf.shm_segsz) ;
     printf("pid du createur: %d\n",buf.shm_cpid) ;
     printf("pid  (derniere operation): %d\n",buf.shm_lpid) ;

     /* destruction du segment */
     if ((shmctl(shmid, IPC_RMID, NULL)) == -1){
          perror("Erreur shmctl()") ;
          exit(1) ;
     }
}
```

Résultat de l'exécution:

```
bolz> ipcs
IPC status from /dev/kmem as of Mon Jun 24 08:57:42 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m   7168 0x7b1499a6 -Crw-------     bolz systemeV
Semaphores:
bolz> test_shmctl
ETAT DU SEGMENT DE MEMOIRE PARTAGEE 7168
identificateur de l'utilisateur proprietaire: 452
identificateur du groupe proprietaire: 226
identificateur de l'utilisateur createur: 452
identificateur du groupe createur: 226
mode d'acces: 33664
taille de la zone memoire: 1000
pid du createur: 23124
pid  (derniere operation): 0
bolz> ipcs
IPC status from /dev/kmem as of Mon Jun 24 08:58:27 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
```

8.6 Primitive shmat()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat(shmid, shmaddr, shmflg)
int shmid ;
char *shmaddr ;
int shmflg ;
```

Valeur retournée: adresse du segment de mémoire partagée, ou -1 en cas d'erreur.

Avant qu'un processus puisse utiliser un segment de mémoire partagée, il doit tout d'abord attacher ce segment à lui-même grâce à la primitive `shmat()`. Celle-ci prend trois arguments: l'identificateur du segment *shmid*, un pointeur vers un caractère *shmaddr*, et un drapeau *shmflg*.
Le segment est attaché à une adresse déterminée par les paramètres *shmaddr* et *shmflg*:

- si *shmaddr* est nul, le segment est attaché à la première adresse possible déterminée par le système.

- si *shmaddr* n'est pas nul, on s'intéresse à *shmflg*:
  - si le drapeau SHM_RND n'est pas positionné dans *shmflg*, le segment est attaché à l'adresse *shmaddr*.
  - si SHM_RND est positionné, le segment est attaché à l'adresse *shmaddr* – (*shmaddr* modulo SHMLBA), où SHMLBA est une constante définie dans `<sys/shm.h>`.

De plus, si le drapeau SHM_RDONLY est positionné dans *shmflg*, le processus ne peut pas écrire dans la zone de mémoire partagée.

Remarques:

Lors de l'appel à `shmat()`, le système vérifie qu'il existe suffisamment de place disponible dans l'espace de mémoire virtuelle du processus auquel on veut attacher le segment de mémoire. Si ce n'est pas le cas, il renvoie une erreur.
Notons bien qu'il n'y a pas copie de zone mémoire, mais simplement redirection de l'adressage vers le segment partagé.

Exemple:

Supposons un segment de mémoire partagée déjà créé. Le programme `test_shmat` rattache un processus au segment, et écrit en mémoire commune une chaine de caractère. Le programme `test_shmat2` se rattache à la même zone mémoire et lit son contenu.

```
                 /* fichier test_shmat.c */

/*
 * exemple d'utilisation de shmat()
 * ecriture dans un segment de memoire partage
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 123

main()
{
  int shmid ;                          /* identificateur de la memoire commune */
  int size = 1000 ;                    /* taille en octet de la memoire */
  char *path="nom_de_fichier_existant" ;
  char *mem ;                          /* pointeur sur la zone commune */
  int flag = 0 ;                       /* drapeau associe au segment */
```

```
    /*
     * recuperation du shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)CLE), size,0)) == -1) {
          perror("Echec de shmget") ;
          exit(1) ;
     }
     printf("identificateur du segment: %d \n",shmid) ;
     printf("ce segment est associe a la cle unique: %d\n",
                   ftok(path,(key_t)CLE)) ;

    /*
     * attachement du processus a la zone de memoire
     * recuperation du pointeur sur la zone memoire commune
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
          perror("attachement impossible") ;
          exit (1) ;
     }

    /*
     * ecriture dans la zone de memoire partagee
     */
     strcpy(mem,"Ceci est ecrit en memoire commune") ;
}
```

```
                 /* fichier test_shmat2.c */

/*
 * ce programme permet de lire le contenu d'un segment de memoire
 * partage qui a ete rempli prealablement
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 123

main()
{
  int shmid ;  /* identificateur de la memoire commune */
  int size = 1000 ;
```

```
  char *path="nom_de_fichier_existant" ;
  char *mem ;
  int flag = 0 ;

    /*
     * recuperation du shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)CLE), size,0)) == -1) {
          perror("Echec de shmget") ;
          exit(1) ;
     }
     printf("identificateur du segment: %d \n",shmid) ;
     printf("ce segment est associe a la cle unique: %d\n",
                  ftok(path,(key_t)CLE)) ;

    /*
     * attachement du processus a la zone de memoire
     * recuperation du pointeur sur la zone memoire commune
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
          perror("attachement impossible") ;
          exit (1) ;
     }

    /*
     * traitement du contenu du segment
     */
     printf("lecture du segment memoire partage : %s\n",mem) ;
}
```

Résultat de l'exécution:

```
$ ipcs
IPC status from /dev/kmem as of Mon Jun 24 09:41:12 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m   8192 0x7b1499a6 --rw-------     bolz systemeV
Semaphores:
$ test_shmat
identificateur du segment: 8192
ce segment est associe a la cle unique: 2064947622
$ test_shmat2
identificateur du segment: 8192
ce segment est associe a la cle unique: 2064947622
lecture du segment memoire partage : Ceci est ecrit en memoire commune
```

8.7 Primitive shmdt()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt(shmaddr)
char *shmaddr ;
```

Valeur retournée: 0 en cas de réussite, ou -1 en cas d'erreur.

Cette primitive permet de détacher un segment de mémoire partagée d'un processus.
Elle prend un seul argument: l'adresse (donnée par *shmaddr*) du segment qui doit être détaché du processus appelant. Celui-ci ne pourra alors plus utiliser cette zone mémoire.

Exemple:

```
                 /* fichier test_shmdt.c */

/*
 * ce programme permet de lire le contenu d'un segment de memoire
 * partage qui a ete rempli prealablement. Le processus apres sa
 * lecture se detache du segment
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define CLE 123

main()
{
  int shmid ;  /* identificateur de la memoire commune */
  int size = 1000 ;
  char *path="nom_de_fichier_existant" ;
  char *mem ;
  int flag = 0 ;

    /*
     * recuperation du shmid
     */
     if (( shmid = shmget(ftok(path,(key_t)CLE), size,0)) == -1) {
          perror("Echec de shmget") ;
          exit(1) ;
```

```
     }
     printf("identificateur du segment: %d \n",shmid) ;
     printf("ce segment est associe a la cle unique: %d\n",
                  ftok(path,(key_t)CLE)) ;

    /*
     * attachement du processus a la zone de memoire
     * recuperation du pointeur sur la zone memoire commune
     */
     if ((mem = shmat (shmid, 0, flag)) == (char*)-1){
          perror("attachement impossible") ;
          exit (1) ;
     }

    /*
     * traitement du contenu du segment
     */
     printf("lecture du segment memoire partage : %s\n",mem) ;

    /*
     * detachement du segment
     */
     if (shmdt(mem)== -1){
          perror("detachement impossible") ;
          exit(1) ;
     }

    /*
     * destruction du segment
     */
     if ( shmctl(shmid, IPC_RMID,0) == -1){
          perror("destruction impossible") ;
          exit(1) ;
     }
}
```

Résultat de l'exécution:

```
$ ipcs
IPC status from /dev/kmem as of Mon Jun 24 10:03:05 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m   8192 0x7b1499a6 --rw-------     bolz systemeV
Semaphores:
$ test_shmat
identificateur du segment: 8192
ce segment est associe a la cle unique: 2064947622
```

```
$ test_shmdt
identificateur du segment: 8192
ce segment est associe a la cle unique: 2064947622
lecture du segment memoire partage : Ceci est ecrit en memoire commune
$ ipcs
IPC status from /dev/kmem as of Mon Jun 24 10:03:50 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
```

Chapter 9

LES MESSAGES

9.1 Introduction

La communication inter-processus par messages s'effectue par échange de données, stockées dans le noyau, sous forme de files. Chaque processus peut émettre des messages et en recevoir.

9.2 Principe

De même que pour les sémaphores et pour la mémoire partagée, une file de messages est associée à une clé qui représente son nom numérique. Cette clé est utilisée pour définir et obtenir l'identificateur de la file de messages, noté *msqid*. Celui-ci est fourni par le système au processus qui donne la clé associée.

Un processus qui désire envoyer un message doit obtenir l'identificateur de la file *msqid*, grâce à la primitive `msgget()`. Il utilise alors la primitive `msgsnd()` pour obtenir le stockage de son message (auquel est associé un type), dans une file.

De la même manière, un processus qui désire lire un message doit se procurer l'identificateur de la file par l'intermédiaire de la primitive `msgget()`, avant de lire le message en utilisant la primitive `msgrcv()`.

9.3 Structure associée aux messages: msqid_ds

Comme nous l'avons vu, chaque file de messages est associée à un identificateur *msqid*. On lui associe également une structure, définie dans le fichier `<sys/msg.h>`, définie comme suit:

```
struct msqid_ds
{ /* une pour chaque file dans le systeme */
        struct ipc_perm msg_perm;       /* operations permises */
        struct msg      *msg_first;     /* pointeur sur le premier message
                                         * d'une file */
```

```
        struct msg      *msg_last;      /* pointeur sur le dernier mesaage
                                         * d'une file */
        ushort          msg_cbytes;     /* nombre courant d'octets de la file */
        ushort          msg_qnum;       /* nombre de message dans la file */
        ushort          msg_qbytes;     /* nombre maximal d'octets de la file */
        ushort          msg_lspid;      /* pid du dernier processus ecrivain */
        ushort          msg_lrpid;      /* pid du dernier processus lecteur */
        time_t          msg_stime;      /* date de la derniere ecriture */
        time_t          msg_rtime;      /* date de la derniere lecture */
        time_t          msg_ctime;      /* date du dernier changement */
};
```

9.4 Primitive msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key, msgflg)
key_t key ;
int msgflg ;
```

Valeur retournée: l'identificateur *msqid* de la file , ou -1 en cas d'erreur.

Cette primitive est utilisée pour créer une nouvelle file de messages, ou pour obtenir l'identificateur de file *msqid* d'une file de messages existante.
Elle prend deux arguments. Le premier (*key*) est une clé indiquant le nom numérique de la file de messages. Le second (*msgflg*) est un drapeau spécifiant les droits d'accès sur la file.

9.4.1 Les valeurs possibles pour *key*

- IPC_PRIVATE (=0): la file de messages n'a alors pas de clé d'accès, et seul le processus propriétaire, ou le créateur a accès à cette file.

- la valeur désirée de la clé de la file de messages.

9.4.2 Les valeurs possibles pour *msgflg*

On remarque que *msgflg* est semblable à *semflg*, utilisé pour les sémaphores, et à *shmflg*, utilisé pour la mémoire partagée.
Ce drapeau est la combinaison de différentes constantes prédéfinies, permettant d'établir les droits d'accès, et les commandes de contrôle (la combinaison est effectuée de manière classique à l'aide de OU '|').

Les constantes prédéfinies dans `<sys/msg.h>`, et `<sys/ipc.h>` sont:

```
#define   MSG_R   400   /* permission en lecture pour l'utilisateur */
#define   MSG_W   200   /* permission en lecture pour l'utilisateur */

#define   IPC_CREAT   0001000   /* creation d'une file de messages */
#define   IPC_EXCL    0002000   /* associe au bit IPC_CREAT provoque */
                                /* un echec si le fichier existe deja */
```

9.4.3 Comment créer une file de messages

La création d'une file de messages est semblable à la création d'un ensemble de sémaphores ou d'un segment de mémoire partagée. Il faut pour cela respecter les points suivants:

- ***key*** **doit contenir la valeur identifiant la file.**
- ***msgflg*** **doit contenir les droits d'accès désirés et la constante IPC_CREAT.**
- **si l'on désire tester l'existence d'une file correspondant à la clé désirée, il faut rajouter à** ***msgflg*** **la constante IPC_EXCL. L'appel** `msgget()` **échouera dans le cas d'existence d'une telle file.**

Notons que lors de la création d'une file de messages, un certain nombre de champs de l'objet de structure `msqid_ds` **sont initialisés (propriétaire, modes d'accès).**

Exemple d'utilisation de msgget():

Ce programme crée une file de messages associée à la clé 123, et vérifie le contenu des structures du système propres à cette file.

```
                 /* fichier test_msgget.c */

/*
 *  exemple d'utilisation de msgget()
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 123

main()
{
  int msqid ;  /* identificateur de la file de messages */
  char *path = "nom_de_fichier_existant" ;

    /*
```

```
    * creation d'une file de messages en lecture et ecriture
    * si elle n'existe pas
    */
    if (( msqid = msgget(ftok(path,(key_t)CLE),
                    IPC_CREAT|IPC_EXCL|MSG_R|MSG_W)) == -1) {
         perror("Echec de msgget") ;
         exit(1) ;
    }
  printf("identificateur de la file: %d\n",msqid) ;
  printf("cette file est identifiee par la cle unique : %d\n"
                         ,ftok(path,(key_t)CLE)) ;
}
```

Résultat de l'exécution:

```
identificateur de la file: 700
cette file est identifiee par la cle unique : 2064941749
```

9.5 Primitive msgctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(msqid, cmd, buf) ;
int msqid, cmd ;
struct msqid_ds *buf ;
```

Valeur retournée: 0 en cas de réussite, -1 sinon.

L'appel système `msgctl()` est utilisé pour examiner et modifier des attributs d'une file de messages existante. Il prend trois arguments: un identificateur de file de messages (*msqid*), un paramètre de commande (*cmd*), et un pointeur vers une structure de type `msqid_ds` (*buf*).

Les différentes commandes possibles

Les commandes sont définies dans le fichier `<sys/ipc.h>`:

- **IPC_RMID (0):** La file de messages identifiée par *msqid* est détruite. Seul le super-utilisateur ou un processus ayant pour numéro d'utilisateur `msg_perm.uid` peut détruire une file. Toutes les opérations en cours sur cette file échoueront et les processus en attente de lecture ou d'écriture sont réveillés.

- **IPC_SET (1):** Donne à l'identificateur de groupe, à l'identificateur d'utilisateur, aux droits d'accès de la file de messages, et au nombre total de caractères des textes, les valeurs contenues dans le champ `msg_perm` de la structure pointée par *buf*. On met également à jour l'heure de modification.

- **IPC_STAT (2): La structure associée à *msqid* est rangée à l'adresse pointée par *buf*.**

Exemple:

Dans cet exemple, on suppose que le segment de mémoire partagée de clé 123 a été préalablement créé.

```
                 /* fichier test_msgctl.c */

/*
 * le programme recupere l'identificateur d'une file existante (creee
 * avec test_msgget.c) et affiche la structure msqid_ds associee a la file
 */

#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 123

main()
{
  struct msqid_ds buf ;
  char *path = "nom_de_fichier_existant" ;
  int msqid ;
     /* recuperation de l'identificateur de la file de messages de cle 123 */
     if (( msqid = msgget(ftok(path,(key_t)CLE),0)) == -1 ) {
          perror ("Erreur msgget()") ;
          exit(1) ;
      }
      /* on recupere dans la structure buf les parametres de al file */
      if (msgctl(msqid,IPC_STAT,&buf) == -1){
          perror("Erreur msgctl()") ;
          exit(1) ;
     }
     else
     {
          printf("id de la file de messages        : %d\n",msqid) ;
          printf("id du proprietaire               : %d\n",buf.msg_perm.uid) ;
          printf("id du groupe du proprietaire     : %d\n",buf.msg_perm.gid) ;
          printf("id du createur                   : %d\n",buf.msg_perm.cuid) ;
          printf("id du groupe du createur         : %d\n",buf.msg_perm.cgid) ;
          printf("droits d'acces                   : %d\n",buf.msg_perm.mode) ;
          printf("nb courant d'octets dans la file : %d\n",buf.msg_cbytes) ;
          printf("nb de messages dans la file      : %d\n",buf.msg_qnum) ;
          printf("nb maximal d'octets de la file   : %d\n",buf.msg_qbytes) ;
```

```
          printf("pid du dernier ecrivain          : %d\n",buf.msg_lspid) ;
          printf("pid du dernier lecteur           : %d\n",buf.msg_lrpid) ;
          printf("date de la derniere ecriture : %s\n",ctime(&buf.msg_stime)) ;
          printf("date de la derniere lecture  : %s\n",ctime(&buf.msg_rtime)) ;
          printf("date du dernier changement   : %s\n",ctime(&buf.msg_ctime)) ;

     }
}
```

Résultat de l'exécution:

```
id de la file de messages        : 700
id du proprietaire               : 452
id du groupe du proprietaire     : 226
id du createur                   : 452
id du groupe du createur         : 226
droits d'acces                   : 33152
nb courant d'octets dans la file : 0
nb de messages dans la file      : 0
nb maximal d'octets de la file   : 16384
pid du dernier ecrivain          : 0
pid du dernier lecteur           : 0
date de la derniere ecriture : Wed Dec 31 18:00:00 1969

date de la derniere lecture  : Wed Dec 31 18:00:00 1969

date du dernier changement   : Mon Jun 24 11:19:01 1991
```

9.6 Primitive msgsnd()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(msqid, msgp, msgsz, msgflg) ;
int msqid ;
struct msgbuf *msgp ;
int msgsz, msgflg ;
```

Valeur retournée: 0 si le message est placé dans la file, -1 en cas d'erreur.

Cette primitive permet de placer un message dans une file.

Elle prend quatre arguments: l'identificateur de la file (*msqid*), un pointeur (*msgp*) vers la structure de type `msgbuf` **qui contient le message, un entier (*msgsz*) indiquant la taille (en otets) de la partie texte du message, et un drapeau (*msgflg*) agissant sur le mode d'exécution de l'envoi du message.**

La primitive `msgsnd()` met à jour la structure `msqid_ds`:

- incrémentation du nombre de messages de la file (`msg_qnum`)
- modification du numéro du dernier écrivain (`msg_lspid`)
- modification de la date de dernière écriture (`msg_stime`)

La structure msgbuf

La structure `msgbuf` décrit la stucture du message proprement dit. Elle est définie dans `/usr/include/sys/msg.h` de la façon suivante:

```
struct msgbuf {
     long mtype ;       /* type du message */
     char mtext[1] ;    /* texte du message */
}
```

mtype est un entier positif. On peut s'en servir pour effectuer en lecture une sélection parmi les éléments présents dans la file.
Il est impératif que le champ*mtype* soit au début de la structure.
mtext est le message envoyé (tableau d'octets). Bizarrement, sa taille est limitée à 1 octet dans `msg.h`. Comme il est couramment nécessaire d'utiliser des messages d'une longueur supérieure à 1, on pourra, par exemple, définir une autre structure (que l'on utilisera à la place de `msgbuf`), de la façon suivante:

```
#define MSG_SIZE_TEXT 256

struct msgtext {
     long mtype ;     /* type du message */
     char mtexte[MSG_SIZE_TEXT] ;    /* texte du message */
} ;
```

On pourra régler, suivant ses besoins, la taille maximum des messages échangés avec `MSG_SIZE_TEXT`.
Il faut noter que le champ *mtype* est bien placé au début de la structure.

Drapeau *msgflg*

Pour ce qui concerne le drapeau *msgflg*, il est utilisé comme suit:

- ce paramètre est mis à 0 pour provoquer le blocage de `msgsnd()` lorsque la file de messages est pleine.
- il est positionné à IPC_NOWAIT pour retourner immédiatement de `msgsnd()` avec une erreur lorsque la file est pleine.

Cet indicateur agit comme O_NDELAY sur les conduits nommés.

Exemple d'utilisation de la primitive `msgsnd()`:

```
                    /* fichier test_msgsnd.c */
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 123
#define MSG_SIZE_TEXT 256

main()
{
int i = 1 ;
int msqid ;               /* identificateur de la file de message */
char *path = "nom_de_fichier_existant" ;
struct msgtext {
     long mtype ;
     char mtext[MSG_SIZE_TEXT] ;
} msg ;                   /* msg structure associee au message */

     /* recuperation de l'identificateur de la file de messages */
     if (( msqid = msgget(ftok(path,(key_t)CLE),0)) == -1 ) {
          perror ("Erreur msgget()") ;
          exit(1) ;
     }
     msg.mtype = 1 ;                        /* type des messages */
     while(1)
     {
          sprintf(msg.mtext,"message no %d de type 1",i) ; /* texte */
          /* on envoie le message a la file */
          if(msgsnd(msqid,&msg,strlen(msg.mtext),IPC_NOWAIT) == -1)
          {
               perror("impossible d'envoyer le message") ;
               exit(-1) ;
          }
          printf("message no %d de type %d envoye a %d",i,msg.mtype,msqid) ;
          printf("  texte du message: %s\n",msg.mtext) ;
          i++ ;
     }
}
```

Résultat de l'exécution:

```
message no 1 de type 1 envoye a 700  texte du message: message no 1 de type 1
message no 2 de type 1 envoye a 700  texte du message: message no 2 de type 1
message no 3 de type 1 envoye a 700  texte du message: message no 3 de type 1
```

....

```
message no 39 de type 1 envoye a 700  texte du message: message no 39 de type 1
message no 40 de type 1 envoye a 700  texte du message: message no 40 de type 1
impossible d'envoyer le message: No more processes
```

Le programme envoie dans la file de messages, créée au préalable avec `test_msgget.c`, autant de messages qu'il lui est possible. Le positionnement du drapeau `IPC_NOWAIT` empêche le blocage du processus lorsque la file est pleine. On peut alors regarder avec `test_msgctl.c` l'état de la file:

```
id de la file de messages        : 700
id du proprietaire               : 452
id du groupe du proprietaire     : 226
id du createur                   : 452
id du groupe du createur         : 226
droits d'acces                   : 33152
nb courant d'octets dans la file : 911
nb de messages dans la file      : 40
nb maximal d'octets de la file   : 16384
pid du dernier ecrivain          : 25997
pid du dernier lecteur           : 0
date de la derniere ecriture : Tue Jun 25 03:59:10 1991

date de la derniere lecture  : Wed Dec 31 18:00:00 1969

date du dernier changement   : Tue Jun 25 03:58:44 1991
```

Les valeurs du *pid* du dernier lecteur et de la date de la dernière lecture sont fixées à des valeurs "bidons".

9.7 Primitive msgrcv()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) ;
int msqid ;
struct msgbuf *msgp ;
int msgsz ;
long msgtyp ;
int msgflg ;
```

Valeur retournée: nombre d'octets du message extrait, ou -1 en cas d'erreur.

Cette primitive permet de lire un message dans une file.

Elle prend cinq arguments: l'identificateur de la file (*msqid*), un pointeur (*msgp*) vers la structure de type `msgbuf` qui contient le message, un entier (*msgsz*) indiquant la taille maximum du message à recevoir, un entier (*msgtyp*) indiquant quel message on désire recevoir, et un drapeau (*msgflg*) agissant sur le mode d'exécution de l'envoi du message.

La primitive range le message lu dans une structure pointée par *msgp* qui contient les éléments suivants:

```
long mtype ;              /* type du message */
char mtext[] ;            /* texte du message */
```

La taille du champ *mtext* est fixée selon les besoins (voir `msgsnd()` pour plus de détail).

Pour ce qui concerne le paramètre *msgflg*:

- si IPC_NOWAIT est positionné, l'appel à `msgrcv()` retourne immédiatement avec une erreur, lorsque la file ne contient pas de message de type désiré.
- si IPC_NOWAIT n'est pas positionné, il y a blocage du processus appelant `msgrcv()` jusqu'à ce qu'il y ait un message du type *msgtyp* dans la file.
- si MSG_NOERROR est positionné, le message est tronqué à *msgsz* octets, la partie tronquée est perdue, et aucune indication de la troncature n'est donnée au processus.
- si MSG_NOERROR n'est pas positionné, le message n'est pas lu, et `msgrcv()` renvoie un code d'erreur.

Le paramètre *msgtyp* indique quel message on désire recevoir:

- Si $msgtyp = 0$, on reçoit le premier message de la file, c'est-à-dire le plus ancien.
- Si $msgtyp > 0$, on reçoit le premier message ayant pour type une valeur égale à *msgtyp*.
- Si $msgtyp < 0$, on reçoit le message dont le type a une valeur t qui vérifie:
  - t est minimum
  - $t \leq |msgtyp|$

Par exemple, si l'on considère trois messages ayant pour types 100, 200, et 300, le tableau suivant indique le type du message retourné pour différentes valeurs de *msgtyp*:

msgtyp	type du message retourné
0	100
100	100
200	200
300	300
-100	100
-200	100
-300	100

Exemple d'utilisation de la primitive msgrcv():

```
                /* fichier test_msgrcv.c */
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define CLE 123
#define MSG_SIZE_TEXT 256

struct msgtext {
     long mtype ;
     char mtext[MSG_SIZE_TEXT] ;
} msg ;                   /* msg structure associee au message */

main()
{
int lg ;            /* longueur du message recu */
long type = 1 ;     /* type du message que l'on recherche */
int size_msg = 22 ; /* taille maximum  du texte que l'on veut recevoir */
int msqid ;         /* identificateur de la file */
char *path = "nom_de_fichier_existant" ;

     /* recuperation de l'identificateur de la file de messages de cle 123 */
     if (( msqid = msgget(ftok(path,(key_t)CLE),0)) == -1 ) {
          perror ("Erreur msgget()") ;
          exit(1) ;
     }

     /* on lit dans la file tant qu'il y a des messages                    */
     /* si les messages sont plus grands que taille_msg, ils sont tronques */
     /* lorsque la file est vide, le processus ne se bloque pas            */
     while((lg=msgrcv(msqid,&msg,size_msg,type,IPC_NOWAIT|MSG_NOERROR)) != -1)
     {
          printf("texte du message (longueur %d) recu: %s\n",lg,msg.mtext) ;
     }
     perror("pas de message") ;
     exit(-1) ;
}
```

Résultat de l'exécution:

```
texte du message (longueur 22) recu: message no 1 de type 1
texte du message (longueur 22) recu: message no 2 de type 1
```

....

```
texte du message (longueur 22) recu: message no 8 de type 1
texte du message (longueur 22) recu: message no 9 de type 1
texte du message (longueur 22) recu: message no 10 de type
texte du message (longueur 22) recu: message no 11 de type
```

....

```
texte du message (longueur 22) recu: message no 39 de type
texte du message (longueur 22) recu: message no 40 de type
pas de message: No message of desired type
```

Le programme lit dans la file tant qu'il y a des messages du bon type à lire. Lorsque la file est vide, il n'y a pas de blocage grâce à IPC_NOWAIT. D'autre part, il faut noter que les messages sont tronqués (positionnement de MSG_NOERROR) à la taille spécifiée lors de la réception du message. De plus, *lg* est égal à *size_msg*, dans ce cas précis, puisque le message est tronqué.

On peut consulter l'état de la file avec `test_msgctl.c`:

```
id de la file de messages        : 700
id du proprietaire               : 452
id du groupe du proprietaire     : 226
id du createur                   : 452
id du groupe du createur         : 226
droits d'acces                   : 33152
nb courant d'octets dans la file : 0
nb de messages dans la file      : 0
nb maximal d'octets de la file   : 16384
pid du dernier ecrivain          : 25997
pid du dernier lecteur           : 26143
date de la derniere ecriture : Tue Jun 25 03:59:10 1991

date de la derniere lecture  : Tue Jun 25 04:01:01 1991

date du dernier changement   : Tue Jun 25 03:58:44 1991
```

Chapter 10

EXEMPLES

10.1 Rendez-vous de trois processus

Dans l'exemple suivant, on programme le rendez-vous de trois processus à l'aide des sémaphores et de la mémoire partagée. Le premier programme (`rdv1`) crée un ensemble de deux sémaphores (un pour l'exclusion mutuelle, l'autre pour le rendez-vous), et un segment de mémoire partagée permettant aux processus de connaître à leur arrivée le nombre de processus présents au point de rendez-vous. Le deuxième programme (`rdv2`) simule les processus arrivant en ce point.
Pour l'utiliser, il faut lancer en premier `rdv1` pour réaliser l'initialisation, puis trois fois `rdv2` en background.

```
                /* fichier rdv1.c */

/*-----------------------------------------------------------------------
                     Rendez-vous de n processus
                     --------------------------

               mutex = 1 ; srdv = 0;
               n = 0     ;
               N nbre de processus a attendre

               P(mutex, 1) ;
               n := n + 1 ;

               if (n < N) then

                           begin
                            V(mutex, 1);
                            P(srdv,, 1);
                           end

                          else
```

```
                                begin
                                 V(mutex, 1);
                                 V(srdv, N);
                                end;

------------------------------------------------------------------------*/

/*
 *         INITIALISATION
 * code du processus createur de memoire partagee
 * et des semaphores.
 */

#include <sys/errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

#define     SHMKEY 75 /* cle pour la memoire partagee */
#define     SEMKEY 76 /* cle pour les semaphores */
#define     NBSEM 2   /* nombre de semaphores (mutex et srdv) */

#define K 1024 /* taille segment */

int *pmem;
int shmid, semid;

main()
{
 int i;
 short initarray[NBSEM-1], outarray[NBSEM-1];

    /*
     * creation de la memoire partagee
     */
     if ((shmid = shmget(SHMKEY,K,SHM_R|SHM_W|IPC_CREAT))==-1){
          perror("segment non cree") ;
          exit(1) ;
     }
     pmem = (int *) shmat(shmid, 0, 0);
     pmem[0] = 0; /* nombre de processus arrives (aucun au debut) */
     pmem[1] = 3; /* on attend 3 processus au RDV */

    /*
     * creation des semaphores
     */
```

```
     if ((semid = semget(SEMKEY, NBSEM,SEM_A|SEM_R|IPC_CREAT)) == -1){
          perror(" creation des semaphores non effectuee") ;
          exit(1) ;
     }

    /*
     * initialisation des semaphores
     */
     initarray[0] = 1;         /* mutex = 1 */
     initarray[1] = 0;         /* srdv  = 0 */
     if (semctl(semid, NBSEM-1, SETALL, initarray) == -1){
          perror ("initialisation des sem incorrecte") ;
          exit(1) ;
     }

    /*
     * lecture des parametres des semaphores
     */
     if (semctl(semid, NBSEM-1, GETALL, outarray) == -1){
          perror("lecture des sem incorrecte") ;
          exit(1);
     };

     printf("INITIALISATION\n") ;
     printf(" nombre de processus attendus : %d\n",pmem[1]) ;
     printf(" nombre de processus arrives  : %d\n",pmem[0]) ;
     printf("Pour creer un processus taper la commande : rdv2 &\n");
}


                  /* fichier rdv2.c */

       /*
        *  code execute par les processus pour realiser le rendez-vous
        */

#include     <sys/types.h>
#include     <sys/ipc.h>
#include     <sys/shm.h>
#include     <sys/sem.h>
#include     <sys/errno.h>

#define     SHMKEY     75
#define     SEMKEY     76

#define     K     1024
```

```
int *pmem;
int shmid, semid;
struct sembuf pmutex[1], vmutex[1], psrdv[1], vsrdv[1] ;

main()
{
     int i;

    /*
     * recuperation du shmid
     */
     if ((shmid = shmget(SHMKEY, K, 0)) == -1){
          perror("segment non recupere") ;
          exit(1) ;
     }

    /*
     * recuperation de semid
     */
     if ((semid = semget(SEMKEY, 2, 0)) == -1){
          perror ("semid non recupere") ;
          exit(1) ;
     }

    /*
     * attachement au segment
     */
     if ((pmem = shmat(shmid, 0, 0)) == (int *)-1){
          perror("attachement non effectue") ;
          exit(1) ;
     }

    /*
     * demande de ressource (P(mutex))
     */
     pmutex[0].sem_op = -1;
     pmutex[0].sem_flg = SEM_UNDO;
     pmutex[0].sem_num = 0;
     if (semop(semid, pmutex, 1) == -1){
          perror("p(mutex) non effectue") ;
          exit(1) ;
     }

    /*
     * nombre de processus arrives
     */
     pmem[0] += 1;  /* n = n + 1 */
```

```
printf(" nombre de processus arrives = %d sur %d\n", pmem[0], pmem[1]);


if ( pmem[0] < pmem[1] ){
    /*
     * liberation de ressource (V(mutex))
     */
     vmutex[0].sem_op = 1;
     vmutex[0].sem_flg = SEM_UNDO;
     vmutex[0].sem_num = 0;
     if (semop(semid, vmutex, 1) == -1){
          perror("V(mutex) non effectuee") ;
          exit(1) ;
     }
     printf("je suis le processus %d et je me bloque \n",getpid());

    /*
     * demande de ressource P(srdv)
     */
     psrdv[0].sem_op = -1;
     psrdv[0].sem_flg = SEM_UNDO;
     psrdv[0].sem_num = 1;
     if (semop(semid, psrdv, 1) == -1){
          perror("P(srdv) non effectuee") ;
          exit(1) ;
     }
}
else {
     printf("je suis le processus %d et debloque tous les processus
             bloques\n" ,getpid());

    /*
     * liberation de ressource (V(mutex))
     */
     vmutex[0].sem_op = 1;
     vmutex[0].sem_flg = SEM_UNDO;
     vmutex[0].sem_num = 0;
     if (semop(semid, vmutex, 1) == -1){
          perror("liberation finale non effectuee") ;
          exit(1) ;
     }

    /*
     * liberation de 3 ressources
     */
     vsrdv[0].sem_op = pmem[1];
     vsrdv[0].sem_flg = SEM_UNDO;
```

```
          vsrdv[0].sem_num = 1;
          if (semop(semid, vsrdv, 1) == -1){
               perror("liberation finale non effectuee") ;
               exit(1) ;
          }
     }
     printf("je suis le processus %d et je suis passe au point de rdv\n",
                            getpid());
}
```

Résultat de l'exécution:

```
$ ipcs
IPC status from /dev/kmem as of Mon Jun 24 12:37:46 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
$ rdv1
INITIALISATION
 nombre de processus attendus : 3
 nombre de processus arrives  : 0
Pour creer un processus taper la commande : rdv2 &
$ ipcs
IPC status from /dev/kmem as of Mon Jun 24 12:38:00 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
m  18432 0x0000004b --rw-------     bolz systemeV
Semaphores:
s   1000 0x0000004c --ra-------     bolz systemeV
$ rdv2 &
1908
$  nombre de processus arrives = 1 sur 3
je suis le processus 1908 et je me bloque

$ ps
   PID TTY      TIME COMMAND
  1670 p7       0:00 sh
  1908 p7       0:00 rdv2
  1911 p7       0:00 ps
$ rdv2 &
1922
$  nombre de processus arrives = 2 sur 3
je suis le processus 1922 et je me bloque

$ ps
   PID TTY      TIME COMMAND
```

```
  1670 p7       0:00 sh
  1908 p7       0:00 rdv2
  1922 p7       0:00 rdv2
  1936 p7       0:00 ps
$ rdv2 &
1952
$  nombre de processus arrives = 3 sur 3
je suis le processus 1952 et debloque tous les processus bloques
je suis le processus 1908 et je suis passe au point de rdv
je suis le processus 1952 et je suis passe au point de rdv
je suis le processus 1922 et je suis passe au point de rdv

$ ps
   PID TTY      TIME COMMAND
   132 p7       0:03 tcsh
  1670 p7       0:00 sh
  1958 p7       0:00 ps
```

10.2 Client-serveur

Cet exemple est l'implémentation d'un problème de client-serveur classique utilisant les messages. Deux types de structures peuvent être utilisées:

- Les messages peuvent être échangés au travers de deux files différentes.
- On peut utiliser une seule file, en différenciant les messages à l'aide de deux types distincts.

Dans les deux cas, il faut lancer le programme serveur en background, puis le programme client.

Utilisation de différents types de messages

```
                 /* fichier serveur1.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define  MSGKEY     13      /* cle associee a la file */

#define  MSG_SIZE_TEXT 256  /* longueur de la partie texte des messages */

#define  MSG_SIZEMAX 260    /* MSG_SIZE_TEXT + sizeof(int)     */
                            /* longueur de la partie information des messages */
```

```
#define  TYPE_CLIENT 1       /* type des messages adresses au client  */
#define  TYPE_SERVEUR 2      /* type des messages adresses au serveur */

struct msgform     {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

main()
{
struct msgform msg;                          /* message */
int msgid;                                   /* identificateur de la file */

     /*
      * creation de la file de messages
      */
     if ((msgid = msgget(MSGKEY, MSG_W|MSG_R|IPC_CREAT)) == -1) {
          perror("creation de la file impossible") ;
          exit(1) ;
     }

     for (;;)
     {
          /*
           * reception d'un message
           */
          if (msgrcv(msgid, &msg,MSG_SIZEMAX,TYPE_SERVEUR, 0) == -1) {
               perror("reception impossible") ;
               exit(1) ;
          }

          printf("le serveur %d recoit un message du client %d\n",getpid(),
                                             msg.pid);
          printf("Texte du message recu: %s\n", msg.mtext) ;

          /*
           * envoi d'un message
           */
          msg.mtype  = TYPE_CLIENT ;
          msg.pid = getpid();
          if (msgsnd(msgid, &msg, MSG_SIZEMAX , 0) == -1) {
               perror("envoi impossible") ;
               exit(1) ;
          }
     }
}
```

```
                 /* fichier client1.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define  MSGKEY     13      /* cle associee a la file */

#define  MSG_SIZE_TEXT 256  /* longueur de la partie texte des messages */

#define  MSG_SIZEMAX 260    /* MSG_SIZE_TEXT + sizeof(int)     */
                            /* longueur de la partie information des messages */

#define  TYPE_CLIENT 1       /* type des messages adresses au client  */
#define  TYPE_SERVEUR 2      /* type des messages adresses au serveur */

struct msgform     {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

main()
{
struct msgform msg;                         /* message */
int msgid;                                  /* identificateur de la file */
char *message1 = "je suis le message 1" ;   /* texte du message */

     /*
      * recuperation de l'identificateur de la file de messages
      */
     if ((msgid = msgget(MSGKEY,0)) == -1) {
          perror("recuperation de msgid impossible") ;
          exit(1) ;
     }

     /*
      * envoi d'un message
      */
     strcpy(msg.mtext,message1) ;
     printf("client: envoi du message:%s\n",msg.mtext) ;
     msg.pid= getpid();
```

```
     msg.mtype = TYPE_SERVEUR ;
     if (msgsnd(msgid,&msg,MSG_SIZEMAX,0) == -1) {
          perror("envoi de message impossible") ;
          exit(1) ;
     }

     /*
      * reception d'un message (accuse de reception)
      */
     printf("client: attente de message\n") ;
     if (msgrcv(msgid, &msg, MSG_SIZEMAX, TYPE_CLIENT, 0) == -1) {
          perror("reception de message impossible") ;
          exit(1) ;
     }
     printf("processus client = %d recoit un message du serveur %d\n", getpid(),
                                   msg.pid);
}
```

Utilisation de deux files distinctes

```
                 /* fichier serveur2.c */

#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define  MSGKEY     57      /* cle associee a la file */

#define  MSG_SIZE_TEXT 256  /* longueur de la partie texte des messages */

#define  MSG_SIZEMAX 260    /* MSG_SIZE_TEXT + sizeof(int)     */
                            /* longueur de la partie information des messages */

struct msgform     {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

main()
{
struct msgform msg;         /* message */
int msgid_serv;             /* identificateur de la file du serveur */
int msgid_client;           /* identificateur de la file du client  */
```

```
    /*
     * creation de la file de messages du serveur
     */
    if ((msgid_serv = msgget(MSGKEY, MSG_W|MSG_R|IPC_CREAT)) == -1) {
         perror("creation de la file impossible") ;
         exit(1) ;
    }

    /*
     * creation de la file de messages du client
     */
    if ((msgid_client = msgget(MSGKEY+1, MSG_W|MSG_R|IPC_CREAT)) == -1) {
         perror("creation de la file impossible") ;
         exit(1) ;
    }

    for (;;)
    {
         /*
          * reception d'un message
          */
         if (msgrcv(msgid_serv, &msg,MSG_SIZEMAX, 0, 0) == -1) {
              perror("reception impossible") ;
              exit(1) ;
         }

         printf("le serveur %d recoit un message du client %d\n",getpid(),
                                    msg.pid);
         printf("Texte du message recu: %s\n", msg.mtext) ;

         /*
          * envoi d'un message
          */
         msg.mtype  = 1;
         msg.pid = getpid();
         if (msgsnd(msgid_client, &msg, MSG_SIZEMAX , 0) == -1) {
              perror("envoi impossible") ;
              exit(1) ;
         }
    }
}



                /* fichier client2.c */

#include <errno.h>
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define  MSGKEY     57      /* cle associee a la file */

#define  MSG_SIZE_TEXT 256  /* longueur de la partie texte des messages */

#define  MSG_SIZEMAX 260    /* MSG_SIZE_TEXT + sizeof(int)     */
                            /* longueur de la partie information des messages */

struct msgform     {
     long     mtype;
     int      pid;
     char     mtext[MSG_SIZE_TEXT];
} ;

main()
{
struct msgform msg;         /* message */
int msgid_serv;             /* identificateur de la file du serveur */
int msgid_client;           /* identificateur de la file du client  */
char *message1 = "je suis le message 1" ;    /* texte du message */

     /*
      * recuperation de l'identificateur de la file de messages du serveur
      */
     if ((msgid_serv = msgget(MSGKEY,0)) == -1) {
          perror("recuperation de msgid impossible") ;
          exit(1) ;
     }

     /*
      * recuperation de l'identificateur de la file de messages du client
      */
     if ((msgid_client = msgget(MSGKEY+1,0)) == -1) {
          perror("recuperation de msgid impossible") ;
          exit(1) ;
     }


     /*
      * envoi d'un message
      */
     strcpy(msg.mtext,message1) ;
     msg.pid= getpid();
```

```
     msg.mtype = 1;
     printf("client: envoi du message:%s\n",msg.mtext) ;
     if (msgsnd(msgid_serv,&msg,MSG_SIZEMAX,0) == -1) {
          perror("envoi de message impossible") ;
          exit(1) ;
     }

     /*
      * reception d'un message (accuse de reception)
      */
     printf("client: attente de message\n") ;
     if (msgrcv(msgid_client, &msg, MSG_SIZEMAX, 1, 0) == -1) {
          perror("reception de message impossible") ;
          exit(1) ;
     }
     printf("processus client = %d recoit un message du serveur %d\n", getpid(),
                                   msg.pid);
}
```

Résultat de l'exécution

Les deux méthodes donnent le même résultat:

```
$
$ ipcs
IPC status from /dev/kmem as of Thu Jun 27 10:54:10 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
Shared Memory:
Semaphores:
$ serveur1 &
12688
$ ipcs
IPC status from /dev/kmem as of Thu Jun 27 10:54:22 1991
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:
q    102 0x0000000d -Rrw-------     bolz systemeV
Shared Memory:
Semaphores:
$ client1
client: envoi du message:je suis le message 1
client: attente de message
le serveur 12688 recoit un message du client 12696
Texte du message recu: je suis le message 1
processus client = 12696 recoit un message du serveur 12688
$
```

Part IV

ANNEXE

Le tableau ci-dessous présente les codes d'erreur les plus fréquemment rencontrés, avec les valeurs que prend la variable `errno`, a insi que les messages renvoyés par `perror()`.

Liste des codes et messages d'erreur usuels		
nom	numéro	texte
EPERM	1	Not super-user
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	IO error
ENXIO	6	No such device or address
E2BIG	7	Arg list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No children
EAGAIN	11	No more processes
ENOMEM	12	Not enough core
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Mount device busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe
EDOM	33	Math arg out of domain of func
ERANGE	34	Math result not representable
ENOMSG	35	No message of desired type
EIDRM	36	Identifier removed
EDEADLK	45	Record locking deadlock
ENOLCK	46	No record locks available

BIBLIOGRAPHIE

- **UNIX programmation avancée, M. J. ROCHKIND (traduction) (Masson)**
- **UNIX Network Programming, W. R. STEVENS (Prentice Hall)**
- **Using C on the UNIX system, D. A. CURRY (O'Reilly Associates)**
- **La programmation sous UNIX, J-M.RIFFLET (Mc Graw-Hill)**
- **Notes sur l'implémentation du noyau UNIX, E. PAIRE**
- **Communications inter-processus sous UNIX système V, C. SEMUR & J-F. PUJOL**