IMT Mines Alès
École Mines-Télécom

Informatique concurrente et répartie

Les processus lourds

Emmanuel Romagnoli

Plan

- Les processus, les groupes de processus et les sessions
- La création et l'arrêt d'un processus lourd
- La synchronisation des processus lourds
- Les objets IPC
- Les signaux
- Les tubes
- Les sockets UNIX

Les processus, les groupes de processus et les sessions

Les processus

Chaque processus se voir attribuer un identifiant (Process IDentifier ou **PID**) qui peut être utilisé dans des instructions comme kill pour envoyer des signaux.

Le premier processus est « init » et a comme PID 1. Les autres valeurs de PID sont attribuées de façon incrémentale jusqu'à la valeur spécifiée dans /proc/sys/kernel/pid_max (généralement 32768). On repart alors à 2 en évitant les PID déjà utilisés.

Le PID du processus père est appelé **PPID** ou Parent Process IDentifier.

Les groupes de processus

Il est possible de grouper des processus au sein de « process-group » identifié par un seul **GID** afin d'en faciliter la gestion (par exemple destruction, en une seule instruction, des processus d'un même groupe).

Ce GID correspond en fait au PID du processus leader alors que les autres processus sont qualifiés de membres.

Les processus appartiennent toujours à un groupe. Lorsqu'un processus est créé (commande fork), il devient membre du groupe dont le leader est le père.

Les sessions

Un groupe de processus appartient lui-même à une session identifiée par un **SID** (Session Identifier) qui correspond au PID du processus leader (qui a créé la session).

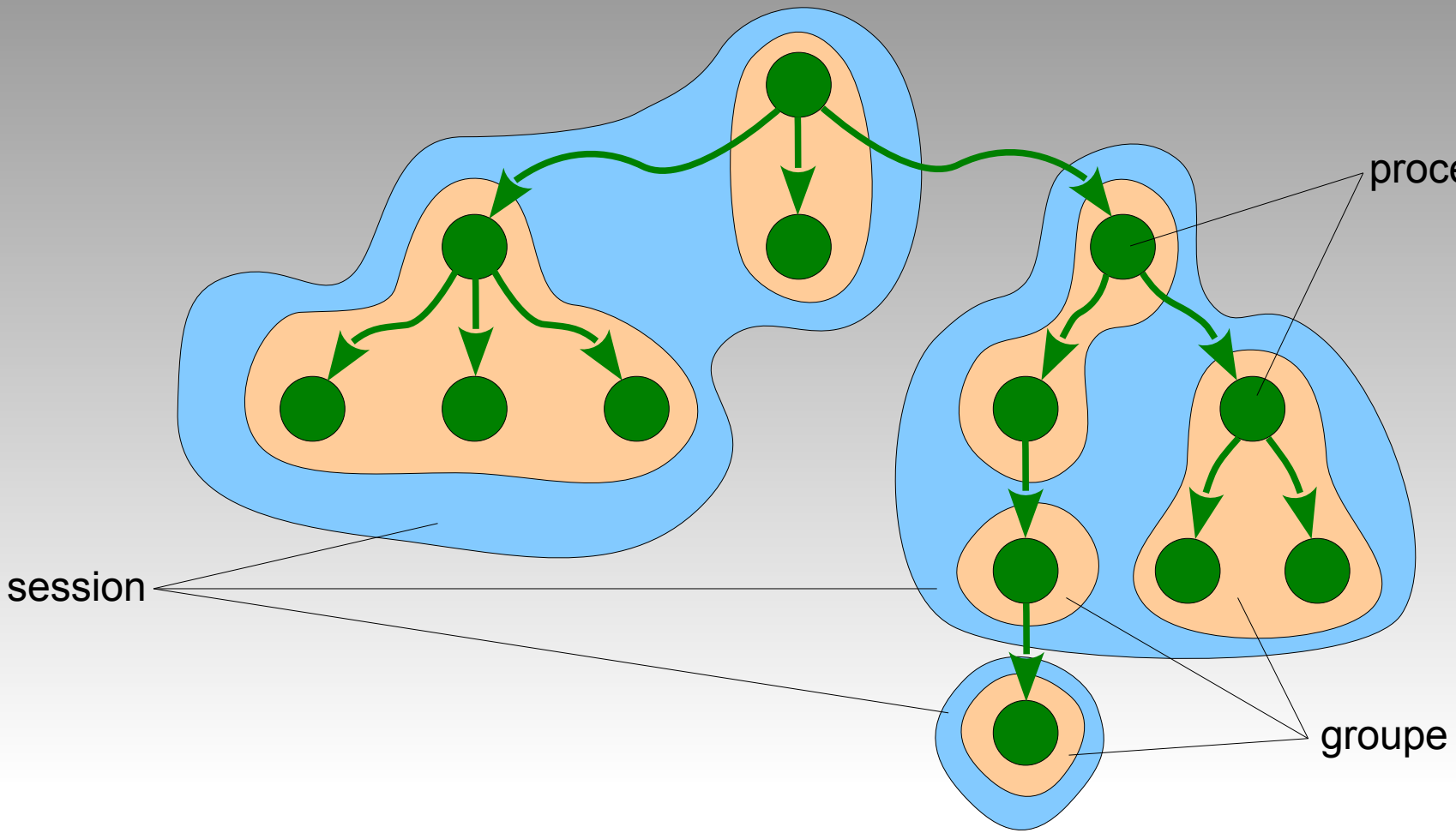
Cette notion de session est utile pour gérer les signaux entre des processus fonctionnant en arrière-plan et d'autres processus chargés de gérer les affichages, les saisies...

Il est également nécessaire de créer des sessions lorsqu'on créé des processus démon, car ils ne doivent être interrompus si leur père vient à disparaître

Voir http://chl.be/glmf/lionel.tricon.free.fr/Articles/Demon/article_demon.pdf
http://www-igm.univ-mlv.fr/~dr/CS/node153.html
http://www.lri.fr/~bastoul/teaching/systeme/docs/TP4_demons.pdf
http://www.editions-eyrolles.com/Chapitres/9782212116014/Chap2-Blaess.pdf?xd=

Synthèse

Les processus ont donc une relation parent-enfant, mais ils appartiennent également à des groupes et à des sessions.



Récupérer du PID et du PPID

La récupération des identifiants est effectuée grâce à deux fonctions définies dans **unistd.h** (il faut aussi inclure **sys/types.h** pour manipuler le type pid_t). :

- **pid_t getpid()** retourne le PID du processus courant ;
- **pid_t getppid()** retourne le PPID du processus courant.

On peut aussi retrouver ces informations depuis le shell en utilisant la commande « **ps -aux** » lorsque les processus sont en cours d'exécution.

Exemple de récupération du PID et du PPID

L'exemple ci-dessous permet d'afficher le PID du processus en cours d'exécution, mais aussi le PID du père (qui correspond à l'interpréteur de commande).

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

int main (int argc, char** argv)
 {
  printf ("Je suis le processus %d\n", getpid());
  printf ("Mon père est le processus %d\n", getppid());
  return 0;
 }
```

Les fonctions de gestion de groupe

La bibliothèque unistd comporte aussi des fonctions qui permettent de gérer les groupes de processus :

- **pid_t getpgid (pid_t pid)** donne le GID du processus dont le PID est passé en paramètre
- **int setpgid (pid_t pid, pid_t gid)** change le GID du processus dont le PID est passé en paramètre (un programme, avec les droits de root, peut changer le GID de tous les processus alors qu'un programme utilisateur ne peut le faire qu'à ses processus fils).

Voir http://pwet.fr/man/linux/appels_systemes/setpgid

Les fonctions de gestion de groupe

setpgid renvoie 0 en cas de succès et -1 en cas d'échec. Dans le second cas, une variable globale **errno** (définie dans **errno.h**) contient le code de l'erreur :

- EINVAL : la valeur de gid est inférieure à 0 donc incorrecte
- EACCES : on a tenté de changer le GID d'un processus fils après qu'il ait exécuté l'instruction execve
- EPERM : on a essayé de placer un processus dans un groupe appartenant à une autre session, de changer le GID d'un processus fils alors que le père se trouve dans une autre session ou de modifier le GID d'un processeur leader d'une session
- ESRCH : la valeur de PID passée en paramètre n'est pas bonne

Les fonctions de gestion de groupe

Il existe aussi d'autres fonctions « raccourcis », mais leur utilisation n'est pas conseillée, car leur portabilité n'est pas garantie :

- **pid_t getpgrp ()** renvoie le GID du processus appelant (équivalent à « getpgid (getpid()) »)
- **int setpgrp ()** crée un groupe en utilisant le processus appelant comme leader (équivalent à « setpgid (0,0) »).

L'obtention d'informations sur un groupe

L'exemple ci-dessous se contente d'afficher le PID, le PPID et le GID. Il nous faut étudier fork pour voir un code permettant de créer nouveau groupe.

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

int main (int argc, char** argv)
 {
  printf ("PID=%d, PPID=%d et GID=%d\n",
           getpid(), getppid(), getpgid (getpid()));

  return 0;
 }
```

Les fonctions de gestion de session

La bibliothèque unistd fournit les deux fonctions suivantes :

- **pid_t getsid(pid_t pid)** renvoie le SID du processus courant si pid est nul ou celui du processus ayant comme PID la valeur fourni en paramètre (si ce PID n'existe pas, la fonction renvoie -1 et errno vaut ESRCH)
- **pid_t setsid(void)** crée une nouvelle session dont le leader est le processus appelant. Si l'appel est un succès, la fonction renvoie le SID (qui est le PID du processus courant) sinon la fonction renvoie -1 et errno est mis à EPERM signifiant que le processus appelant est déjà le leader d'un groupe

Voir http://pwet.fr/man/linux/appels_systemes/setsid
http://pwet.fr/man/linux/appels_systemes/getsid

L'obtention d'informations sur une session

Ce programme se contente d'afficher le PID, le PPID, le GID et le SID. Pour tester la création d'une session, il faut d'abord étudier le fonctionnement de la fonction fork.

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

int main (int argc, char** argv)
 {
  printf ("PID=%d, PPID=%d, GID=%d et SID=%d\n",
           getpid(), getppid(), getpgid (getpid()),
           getsid(0));
  return 0;
 }
```

La création et l'arrêt d'un processus lourd

Le clonage des processus par l'instruction fork

La création d'un nouveau processus s'effectue à l'aide de l'instruction **fork** définie dans **unistd.h**.(il faut aussi inclure sys/types.h pour connaître le type de retour pid_t). Sa signature est la suivante :

```
pid_t fork(void)
```

Un **clone** du processus père est créé et il commence son exécution (lorsqu'il a la main) à l'instruction située juste après le fork.

Les deux processus disposent donc des mêmes variables avec les mêmes valeurs au moment du clonage, mais ces variables évoluent ensuite de façon indépendante.

Les valeurs retournées par fork

L'instruction fork ne prend aucun paramètre, mais retourne un entier qui prend une valeur différente selon la situation :

- il s'agit du processus **fils** : l'instruction fork retourne la valeur **0** ;
- il s'agit du processus **père** : l'instruction fork retourne une valeur strictement supérieure **à 0**, correspondant au **PID du processus fils** ;
- il s'agit de la valeur -1 : le processus fils n'a pas pu être créé et errno contient le numéro de cette erreur.

Voir http://drocourt.info/cours/Unix/C-Programmation_Systeme/prog_sys3.xhtml
http://manpagesfr.free.fr/man/man2/fork.2.html

Les erreurs dans l'exécution du fork

La valeur stockée dans errno peut correspondre à l'une des constantes ci-dessous (définie dans errno.h) :

- EAGAIN : le nombre de processus par utilisateur (défini par la constante CHILD_MAX) est atteint ou le nombre global de processus est à son maximum (table des processus saturée)
- ENOMEM : il n'y a plus assez de mémoire pour le nouveau processus

On peut utiliser la fonction perror, définie dans stdio.h pour afficher un message correspondant à la dernière erreur dont le code est stocké dans errno.

Exemple de création d'un processus

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

int main (int argc, char** argv)
 {
  pid_t pid = fork ();

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n");
              perror ("Erreur : ");
              break;

    case  0 : printf ("On se trouve dans le processus fils.\n");
              printf (" - PID du processus courant : %d\n", getpid());
              printf (" - PID du processus pere    : %d\n", getppid());
              break;

    default : printf ("On se trouve dans le processus pere.\n");
              printf (" - PID du processus courant : %d\n", getpid());
              printf (" - PID du processus fils    : %d\n", pid);
   }

  return 0;
 }
```

L'attaque « Fork bomb »

Cette attaque consiste à créer un programme effectuant des fork au sein d'une boucle. Lors de son exécution, le processus de départ crée d'autres processus qui crée d'autres processus... ce qui a pour effet de saturer la table des processus.

```
#include <unistd.h>

int main (int argc, char** argv)
 {
  while (1)
   fork ();
 }
```

Exemple de création d'un groupe

On reprend l'exemple précédent afin de le compléter.

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

int main (int argc, char** argv)
 {
  pid_t pid = fork ();

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("erreur");
              break;

    case  0 : printf ("Fils : PID=%d, PPID=%d et GID=%d \n",
                       getpid(), getppid(), getpgid (getpid()));
              setpgid(0,0);
              printf ("Fils : PID=%d, PPID=%d et GID=%d \n",
                       getpid(), getppid(), getpgid (getpid())); break;

    default : printf ("Pere : PID=%d, PPID=%d et GID=%d \n",
                       getpid(), getppid(), getpgid (getpid()), getsid(0));
   }
  return 0;  }
```

Exemple de création d'une session

On reprend l'exemple précédent afin de le compléter.

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

int main (int argc, char** argv)
 {
  pid_t pid = fork ();

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("erreur");
              break;

    case  0 : printf ("Fils : PID=%d, PPID=%d, GID=%d et SID=%d\n",
                      getpid(), getppid(), getpgid (getpid()), getsid(0));
              setsid();
              printf ("Fils : PID=%d, PPID=%d, GID=%d et SID=%d\n",
                      getpid(), getppid(), getpgid (getpid()), getsid(0)); break;

    default : printf ("Pere : PID=%d, PPID=%d, GID=%d et SID=%d\n",
                      getpid(), getppid(), getpgid (getpid()), getsid(0));
   }
  return 0;  }
```

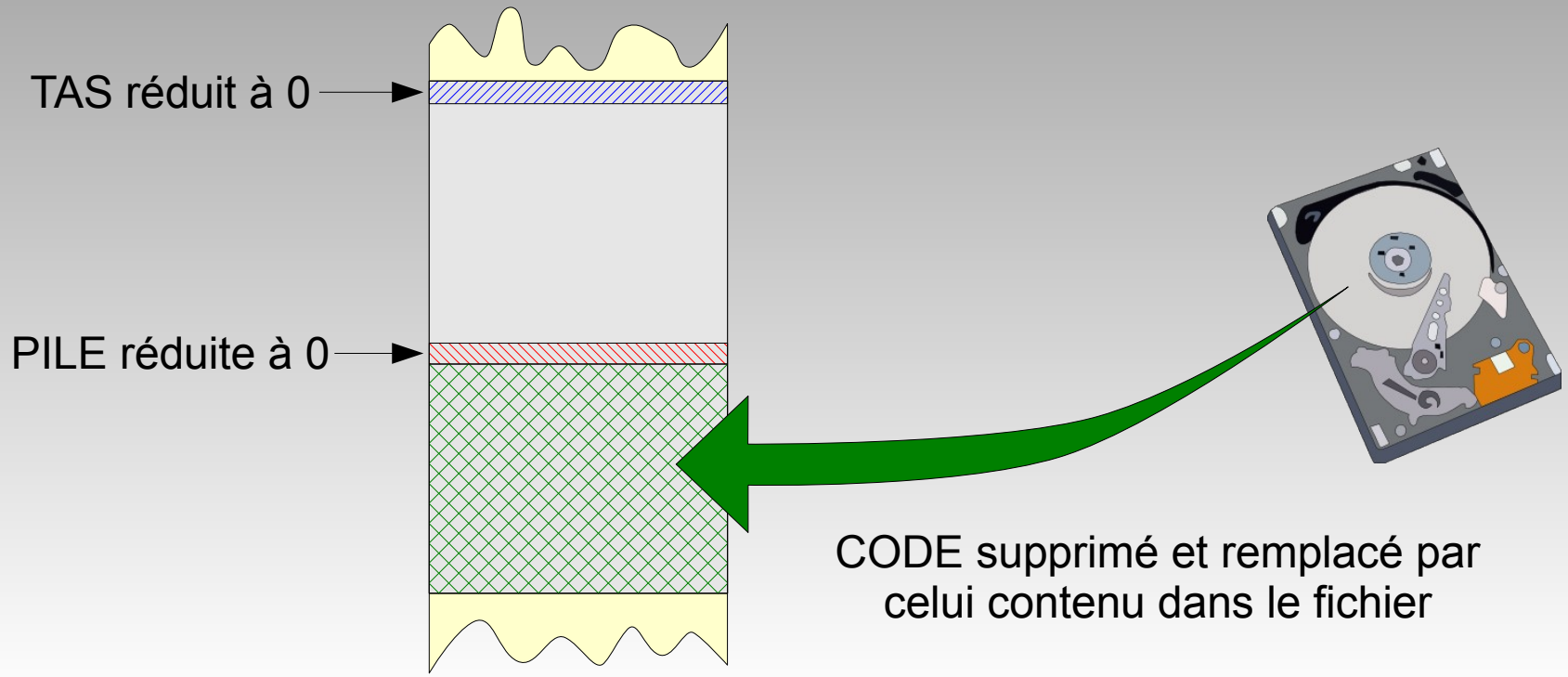
Le lancement d'un programme par le shell

Connaissant la fonction fork et la notion de groupe, on peut maintenant décrire la procédure qui est déroulée lors du lancement d'un programme.

- le shell crée un processus fils grâce à la fonction fork ;
- le fils crée un groupe et en devient le leader en faisant « setpgid(0,0) » ;
- le processus fils appelle une fonction de la famille exec() pour lancer la commande désirée.

La fonction execve

Il s'agit de la fonction de « base » définie dans unistd qui a pour objet de « vider » la zone mémoire affectée au nouveau processus (la copie du shell) pour la remplir avec le code correspondant au programme à exécuter.



Voir http://pubs.opengroup.org/onlinepubs/000095399/functions/exec.html

La fonction execve

La signature de cette fonction est la suivante :

```
int execve (const char *fichier, char* constargv [],
            char* constenvp[]);
```

L'argument fichier peut être :

- le nom complet d'un exécutable binaire ;
- une chaîne de caractères de la forme

  `interpreteur [arg] script`

  (il s'agit d'un script où `fichier` est le chemin vers l'exécutable correspondant à l'interpréteur (la première du script doit être du type « `#! interpreteur [arg]` »).

La fonction execve

constargv est un tableau de chaînes de caractères correspondant aux arguments qui sont passés au programme. Le dernier élément de ce tableau doit être le caractère nul.

```
char *cmd[] = { "ls", "-l", (char *)0 };
```

Le main du programme appelé doit être de la forme :

```
int main (int argc, char** argv)
```

argc contiendra la taille du tableau cmd et argv contiendra le les éléments du tableau constargv, à l'exception du 0 terminal.

La fonction execve

constenvp est un tableau contenant des chaînes de caractères de la forme « nom=valeur ». Il s'agit de variables d'environnements qui sont passées au programme. Le dernier élément de ce tableau doit être le caractère nul.

```
char *env[] = { "HOME=/usr/home",
"JAVA_HOME=/usr/java", (char *)0 };
```

Le main du programme appelé peut être de la forme :

```
int main(int argc, char **argv, char **envp)
```

envp contiendra les éléments du tableau constenvp, à l'exception du 0 terminal. On peut aussi récupérer ces informations grâce à la fonction getenv définie dans stdlib.h.

La fonction execve

Cette fonction renvoie un entier qui vaut 0 en cas de succès et -1 en cas d'échec. Dans le second cas, errno contient le numéro de l'erreur :

- EPERM : le programme doit être exécuté par root
- ENAMETOOLONG : le nom du fichier est trop long
- EMFILE : le nombre de fichiers ouverts trop important
- EACCES : les droits d'accès ne sont pas bons
- E2BIG : constargv et constenvp sont trop grands

Il existe d'autres erreurs possibles concernant cette fonction, il faut donc consulter le manuel pour obtenir des informations complémentaires.

Voir http://linux.die.net/man/2/execve

Exemple de lancement de la commande ls

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

int main (int argc, char** argv)
 {
  int ret;
  char *cmd[] = { "ls", "-l", "/home", (char*) 0 };
  char *env[] = { "HOME=/usr/home", "JAVA_HOME=/usr/java", (char*) 0 };

  pid_t pid = fork ();

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n");
              perror ("Erreur");
              break;

    case  0 : printf ("On se trouve dans le processus fils et on execute ls.\n");
              ret = execve ("/bin/ls", cmd, env);
              break;

    default : printf ("On se trouve dans le processus pere.\n");
   }

  return 0;
 }
```

Les autres fonction exec

Il existe des variantes qui appellent la fonction execve :

- int execl (const char *path, const char *arg, ...)
- int execlp (const char *file, const char *arg, ...)
- int execle (const char *path, const char *arg , ..., char * const envp[])
- int execv (const char *path, char *const argv[])
- int execvp (const char *file, char *const argv[])

Voir http://pubs.opengroup.org/onlinepubs/000095399/functions/exec.html

La fonction system

La fonction system est définie dans stdlib.h et a comme signature :

```
int system (const char * command)
```

Il s'agit d'une sorte de raccourci qui crée un processus fils (fork), lance l'exécution de la commande passée en paramètre (execve), attend la fin de l'exécution du fils (waitpid) et renvoie le résultat (le contenu de la variable d'état). Durant son exécution, les signaux SIGINT et SIGQUIT sont ignorés et le signal SIGCHLD est bloqué.

Voir http://shtroumbiniouf.free.fr/CoursInfo/Systeme2/TP/FctSystUnix/FonctionsC.html#signal%28%29

Les conditions d'un arrêt

Le processus peut se terminer de différentes manières :

- le programme qui est exécuté en son sein est terminé ;
- le processus a exécuté la fonction exit ;
- le processus a reçu un signal d'arrêt.

Les deux premiers cas sont du au fonctionnement interne du processus alors que dans le dernier cas, l'arrêt est une réponse à un stimulus externe qui sera traité plus loin.

La fonction exit

Cette fonction, définie dans la bibliothèque stdlib.h, prend un entier en paramètre qui sera renvoyé vers le processus père. Sa signature est la suivante :

```
void exit (int status)
```

Par convention, un processus se terminant normalement renvoie 0 sinon il renvoie un autre nombre correspondant à un code d'erreur.

Ce nombre peut être récupéré par le processus père grâce à la fonction wait décrite juste après.

Voir http://manpagesfr.free.fr/man/man3/exit.3.html

La synchronisation des processus lourds

L'état zombie

Lorsqu'un processus s'arrête, il envoie un signal SIGCHLD puis il passe dans un état « **zombie** » (defunct) dans lequel quelques la zone de mémoire est libérée alors que l'entrée dans la table des processus est toujours utilisée.

Laisser trop longtemps, trop de processus zombies peut donc empêcher la création de nouveau processus (erreur EAGAIN).

La gestion des processus zombie

Lorsqu'un processus père génère un processus fils, on ne peut pas émettre d'hypothèse quant à leur vitesse d'exécution respective.

On peut avoir deux cas de figure :

- le processus fils s'arrête avant le processus père : le père est chargé de gérer ce processus zombie (en supprimant notamment l'entrée dans la table des processus) ;
- le processus père s'arrête avant le processus fils : le processus fils est alors adopté par le processus init qui est alors responsable de ce processus orphelin (le PPID devient 1).

La fonction wait

Elle nécessite d'inclure les bibliothèques sys/types.h et sys/wait.h. Sa signature est la suivante :

```
pid_t wait(int *status)
```

Elle prend en paramètre un pointeur sur un entier qui contiendra, à l'issue de l'appel à cette fonction, la valeur mise en paramètre de la fonction exit.

L'appel de cette fonction wait provoque la libération de l'entrée dans la table des processus (le PID peut alors être « théoriquement » affecté à un autre processus).

La fonction wait

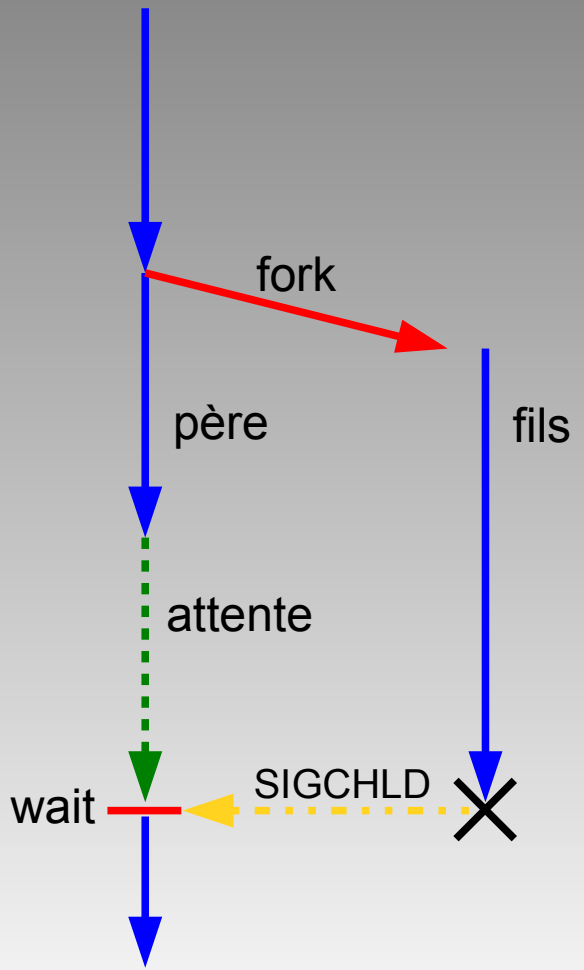
Cette fonction retourne un entier (une valeur de type pid_t) qui peut être :

- une valeur supérieure à 0 correspondant au PID du fils zombie qui a été traité
- -1 qui signifie la survenue de l'erreur ECHILD (le processus appelant n'a aucun fils à attendre).

La barrière de synchronisation

La fonction wait est donc une fonction bloquante pour le processus appelant (le processus père) qui doit attendre qu'un fil passe à l'état zombie (la réception du signal SIGCHLD) pour être débloqué.

Cette fonction joue donc le rôle de barrière de synchronisation entre le père et un processus fils ce qui évite l'apparition de processus orphelin.



Voir http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/book-ora168.html
http://manpagesfr.free.fr/man/man2/wait.2.html

Exemple d'utilisation de wait

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

int main (int argc, char** argv)
 {
  pid_t pid = fork ();
  pid_t pid2;
  int   etat;

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("Erreur");
              break;

    case  0 : printf ("Fils : PID=%d\n", getpid());  sleep (2); /* Attendre 2 secondes */
              printf ("Fils : Fin\n");               exit(0);

    default : printf ("Pere : PID=%d\n", getpid());
              pid2=wait(&etat);
              printf ("Pere : Le fils %d s'est arrete avec le code %d\n", pid2, etat);
              printf ("Pere : Fin\n");
   }

  return 0;
 }
```

La fonction waitpid

Cette fonction a comme signature :

```
pid_t waitpid (pid_t pid, int *status, int options)
```

Il s'agit d'une variante de la fonction wait qui offre plus de possibilités grâce à ses deux nouveaux paramètres :

- pid qui permet de spécifier les processus fils qui doivent être attendus ;
- option qui permet de contrôler le fonctionnement de la fonction waitpid.

La fonction waitpid

L'argument pid peut prendre 4 valeurs :

- Une valeur strictement inférieure à -1 signifie qu'il faut attendre un processus fils dont le GID correspond à la valeur absolue de l'argument pid ;
- -1 signifie qu'il faut attendre un des processus fils ;
- 0 signifie qu'il faut attendre un des processus fils dont le GID du processus est égal à celui du processus père ;
- Une valeur strictement supérieure signifie qu'il faut attendre le processus fils dont le PID est spécifié en argument

La fonction waitpid

L'argument option est un entier issu de OU binaire entre 0 et les constantes ci-dessous :

- WNOHANG qui rend waitpid non bloquant (donc si aucun fil n'est dans l'état zombie, le père n'est pas bloqué) ;
- WUNTRACED qui rend waitpid non bloquant si l'un des fils est bloqué par la réception d'un signal SIGTTIN, SIGTTOU, SIGTSTP ;
- WCONTINUED qui rend waitpid non bloquant si l'un des fils est débloqué par la réception d'un signal SIGCONT.

Voir http://books.google.fr/books?id=VVUbTZnknFIC&pg=PT380#v=onepage&q&f=false

La fonction waitpid

Comme pour la fonction wait, waitpid retourne le PID du fils en cas de succès ou -1 en cas d'échec. Dans le second cas de figure, la valeur de errno peut être la suivante :

- ECHILD : la valeur de pid, spécifiée en argument, ne correspond pas à un processus fils
- EINVAL : la valeur de l'argument option est invalide
- EINTR : l'option WNOHANG n'a pas été activée et un signal non bloqué ou le signal SIGCHLD a été reçu

Des macros complémentaires

L'information inscrite dans la variable status peut être analysée grâce à des instructions qui prennent en paramètre cette valeur :

- WIFEXITED(status) renvoie vrai (1) si le processus fils s'est terminé normalement
- WEXITSTATUS(status) renvoie le code de retour du processus fils (argument de la fonction exit ou de l'instruction return dans le main)
- WIFSIGNALED(status) renvoie vrai (1) si le processus fils s'est terminé après avoir reçu un signal

Des macros complémentaires

- WTERMSIG(status) renvoie le numéro du signal qui entraîné la fin du processus fils
- WCOREDUMP(status) renvoie vrai si le processus a produit un core dump.
- WIFSTOPPED(status) renvoie vrai si le processus est stoppé après reçu un signal (nécessite l'option WUNTRACED ou la surveillance du processus fils avec la fonction ptrace).
- WSTOPSIG(status) retourne le numéro du signal qui a causé l'interruption du processus
- WIFCONTINUED(status) renvoie vrai si le processus fils a redémarré après avoir reçu le signal SIGCONT

Les objets IPC

Le principe

Unix System V a mis en place des objets facilitant la communication entre les processus (IPC étant l'acronyme de Inter Process Communication) :

- la mémoire partagée
- les sémaphores
- les files de messages

Ces objets ont ensuite été repris par d'autres systèmes d'exploitation comme Linux.

Voir http://linux.die.net/man/5/ipc
http://drocourt.info/cours/Unix/C-Programmation_Systeme/prog_sys7.xhtml

L'identification des structures IPC

Ces objets sont partagés par les processus, ils sont donc situés en dehors de ces derniers.

Ces objets sont gérés par le système d'exploitation et sont identifiés grâce à des clés et des descripteurs.

Ces clés permettent aux processus de désigner ces objets lorsqu'ils doivent les manipuler avec les fonctions shmget, semget...

Ces dernières fonctions renvoient généralement des numéros de descripteurs qui sont ensuite utilisés par les processus.

La fonction ftok

Cette fonction est définie dans sys/ipc.h. Elle permet de générer une clé unique à partir :

- d'un chemin d'accès à un fichier (un type char* correspondant à une chaîne de caractères) ;
- d'un entier codé sur 8 bits (un type char).

Cette fonction retourne, en cas de succès, la clé sous la forme d'une variable de type key_t (qui correspond en réalité à un entier). Il est donc nécessaire d'inclure le fichier sys/types.h afin que ce type soit reconnu par le compilateur.

Le prototype de cette fonction est donc la suivante :

```
key_t ftok (char *pathname, char proj)
```

La fonction ftok

En cas d'échec, la fonction retourne -1 et errno peut valoir :

- ENOENT : une partie de la chaîne ne correspond pas à un élément existant.
- ELOOP : il y a trop de liens symboliques dans le chemin d'accès.
- EACCES : les droits d'accès au fichier ne sont pas bons.
- ENAMETOOLONG : le nom du fichier est trop long.

La fonction ftok

Les bénéfices de ce mécanisme de création de la clé sont les suivants :

- Des processus utilisent plusieurs objets partagés : on crée la clé à partir d'un fichier commun (choisi arbitrairement) et d'un numéro différent pour chaque objet.
- Plusieurs utilisateurs font fonctionner les processus évoqués précédemment : le fichier sera alors choisi de manière à être différent d'un utilisateur à l'autre (un fichier commençant par « /home/login » par exemple.

La clé IPC_PRIVATE

IPC_PRIVATE est un « joker » qui permet de demander au système de créer un objet IPC (voir shmget, semget...) sans qu'il soit nécessaire de créer une clé grâce à la fonction ftok.

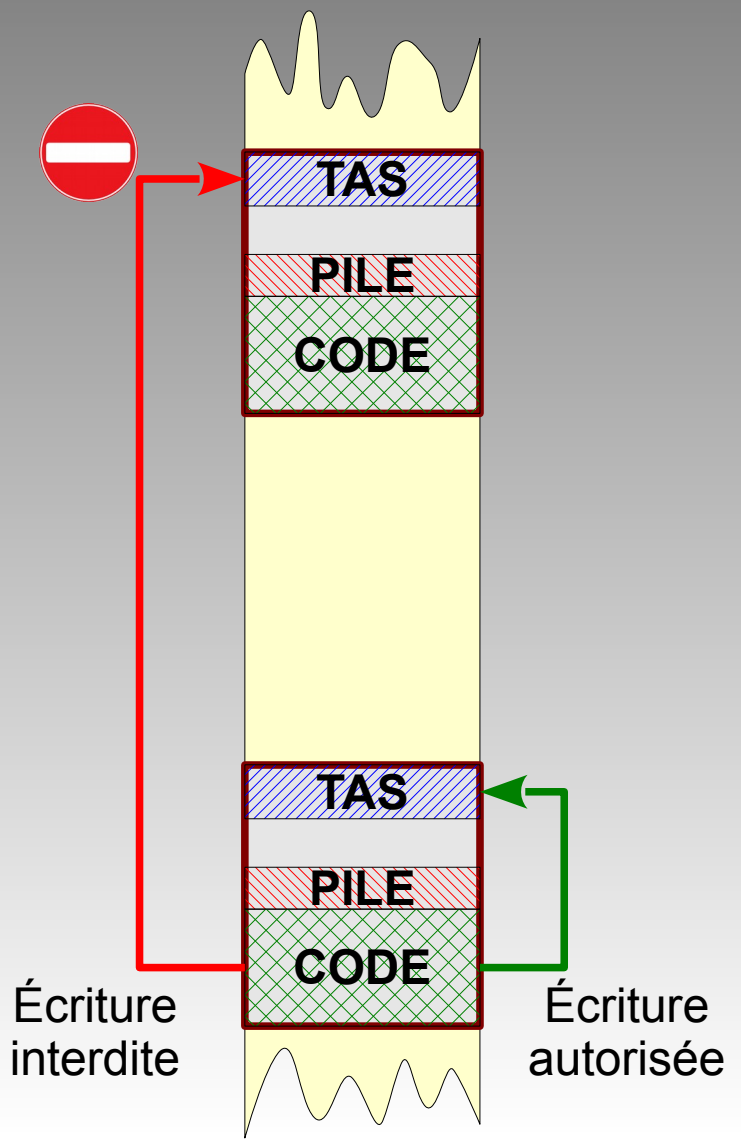
Cela concerne uniquement les objets utilisés par des processus d'une même filiation créés au sein d'application concurrente (instruction fork) : ils possèdent le même numéro de descripteur ce qui est suffisant pour partager l'objet IPC.

Le principe

Chaque processus se voit attribuer une zone de mémoire pour pouvoir fonctionner.

Les mécanismes de protection empêchent d'écrire en dehors de cette zone.

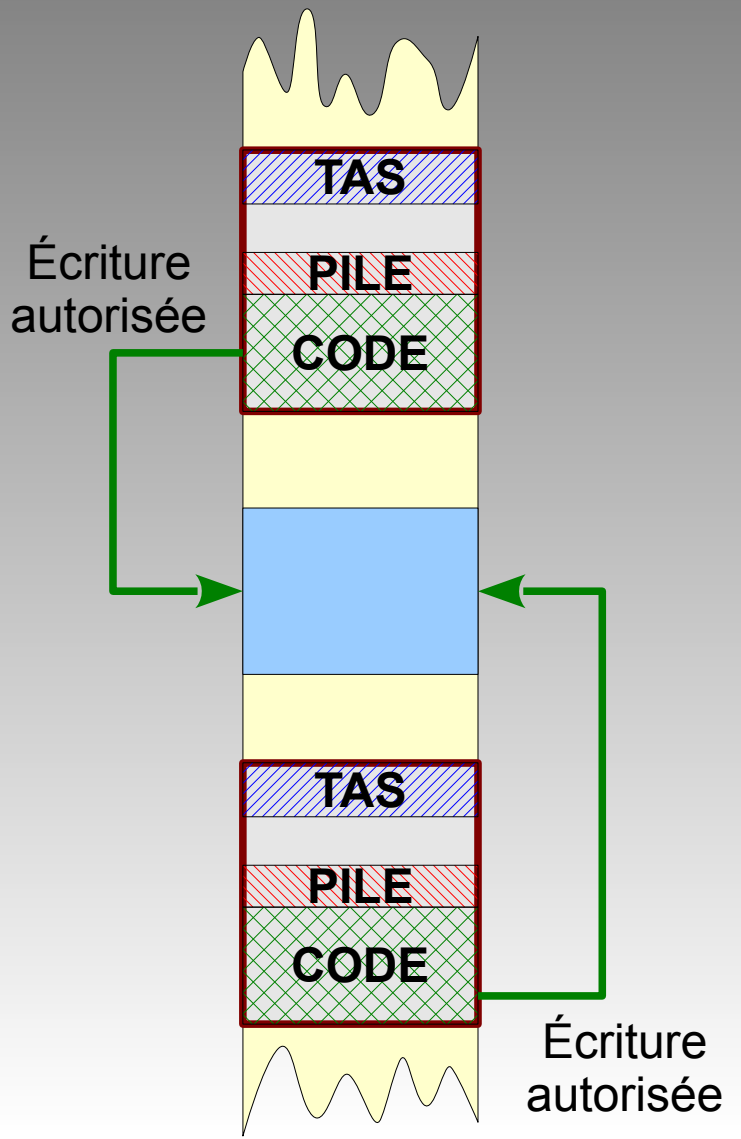
Deux processus, qui veulent communiquer, ne peuvent donc pas écrire directement leur message dans la zone mémoire de leur interlocuteur.



Le principe

Pour contourner ce problème, on alloue une zone de mémoire supplémentaire dans laquelle les deux processus peuvent écrire.

Ils utilisent donc cette zone de mémoire pour communiquer.



La fonction shmget

Comme cette zone de mémoire commune est située en dehors de l'espace mémoire utilisé par le processus, on ne peut pas utiliser une fonction comme malloc pour l'allouer.

On doit demander au système d'exploitation de le faire pour nous en utilisant la fonction **shmget** définie dans sys/shm.h (il faut aussi inclure sys/ipc.h pour accéder à certaines constantes) et dont le prototype est le suivant :

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

La fonction shmget

Le premier argument, de type key_t, est la clé permettant d'identifier l'objet IPC (la mémoire partagée). Cette clé peut être :

- une clé calculée grâce à la fonction ftok (processus sans lien de filiation) ;
- la clé « joker » IPC_PRIVATE (processus possédant un lien de filiation).

Le deuxième argument, de type size_t, est le nombre d'octets qu'il faut allouer.

La fonction shmget

Le dernier argument est un entier qui permet de spécifier les droits d'accès à l'objet IPC et la manière dont la fonction doit fonctionner.

Pour cela, on fait un ou logique entre :

- le nombre représentant les droits d'accès sous la forme 0xyz ;
- les constantes symboliques ci-dessous :
  - IPC_CREAT : création d'un objet IPC s'il n'existe pas ;
  - IPC_EXCL : générer une erreur si l'objet IPC existe.

Rappel concernant les droits d'accès

x est un entier spécifiant les droits d'accès de l'utilisateur. y correspond aux droits d'accès du groupe auquel appartient l'utilisateur. z concerne les droits d'accès des autres utilisateurs (le « reste du monde »).

Chaque entier est codé sur 3 bits :

- le bit 2 est mis à « 1 » si la lecture est autorisée ;
- le bit 1 est mis à « 1 » si l'écriture est autorisée ;
- le bit 0 est mis à « 1 » si l'exécution est autorisée.

Par exemple 762 signifie que l'utilisateur peut tout faire alors que les membres du groupe ne peuvent pas écrire et que les autres utilisateurs ne peuvent que lire.

La fonction shmget

Si l'allocation se déroule correctement, la fonction retourne un numéro de descripteur. Dans le cas contraire, elle renvoie -1 et errno est initialisée à une des valeurs suivantes :

- EACCES : l'utilisateur n'a pas la permission d'accéder au segment de mémoire partagée (créé par un autre utilisateur) et il n'a pas la capacité CAP_IPC_OWNER (capacité des processus privilégiés pour lesquels ils n'y a pas de vérification des droits)
- EEXIST : les options IPC_CREAT et IPC_EXCL ont été activées et le segment existe déjà.

Voir http://marionpatrick.free.fr/man_html/html/capabilities_7.html

La fonction shmget

- EINVAL : un nouveau segment de mémoire partagée a été créé, mais sa taille n'est pas comprise entre SHMMIN et SHMMAX, ou on essaye de créer un segment de mémoire partagée alors qu'il existe déjà avec une taille plus petite (pas possible d'agrandir le segment existant).
- ENFILE : Le nombre total de descripteurs (de fichiers) a été atteint.
- ENOENT : Il n'existe pas de segment correspondant à la clé qui a été donnée en argument et l'option IPC_CREAT n'est pas activée.

La fonction shmget

- ENOMEM : Il n'y a pas assez de mémoire pour allouer le segment de mémoire partagée
- ENOSPC : tous les identifiants de mémoire partagée sont utilisés (voir la constante SHMMNI) ou l'allocation d'un segment de mémoire partagée avec la taille indiquée risque de provoquer un dépassement de capacité pour l'ensemble des segments (voir la constante SHMALL).

L'attachement de la zone de mémoire partagée

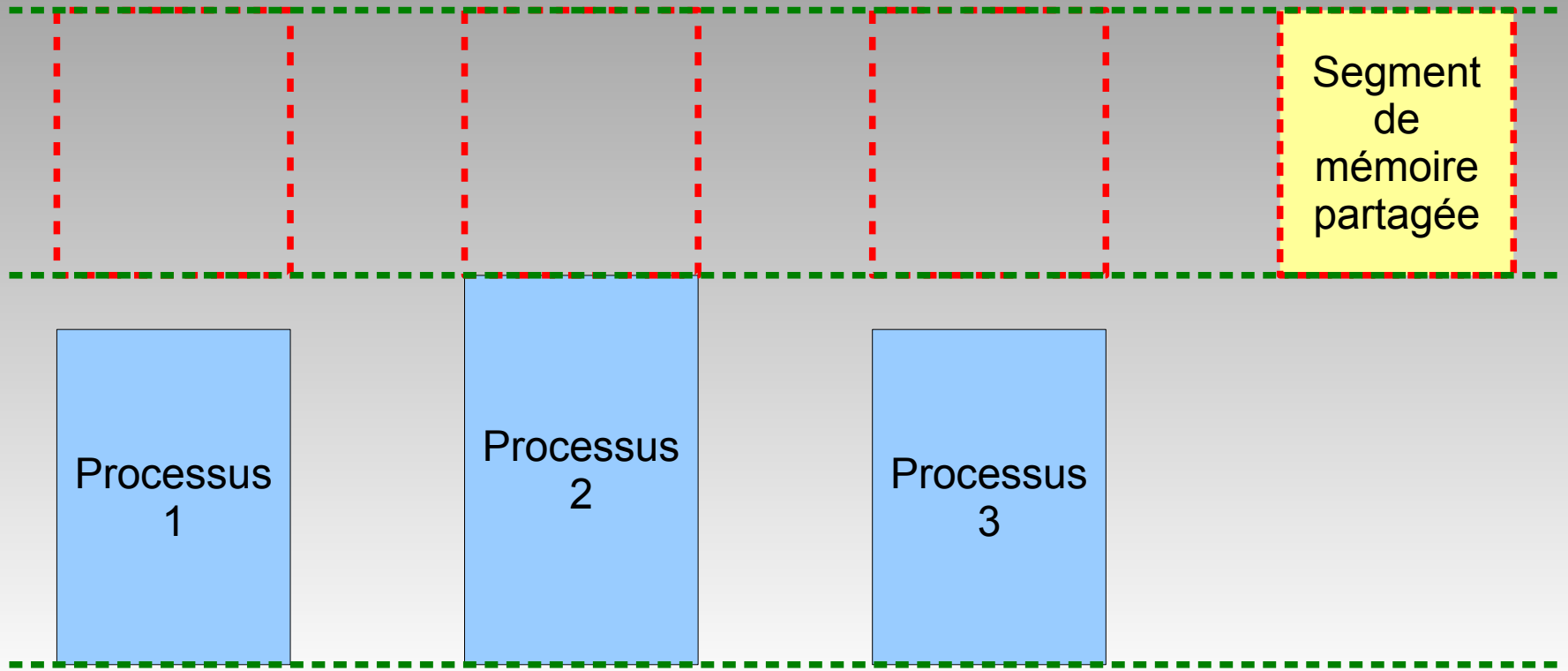
La zone de mémoire a été créée, mais elle n'est pas encore utilisable, car aucune plage d'adresses ne lui a été affectée.

Il faut affecter une plage d'adresses relatives qui ne soient pas en conflit avec les plages d'adresses déjà utilisées par les processus qui accèdent à cette mémoire.

Il faut également que le système d'exploitation conserve une trace de cette affectation afin qu'il puisse réaliser la translation d'adresse lors des accès à cette mémoire

L'attachement de la zone de mémoire partagée

Le schéma de principe ci-dessous représente les processus et la zone de mémoire partagée en utilisant les plages d'adresses relatives.



La fonction shmat

La fonction shmat définie dans sys/shm.h permet de réaliser l'attachement d'une plage d'adresses relatives à une zone de mémoire partagée, allouée par le système d'exploitation (fonction shmget).

Le prototype de cette fonction est le suivante :

```
void *shmat(int shmid, const void *shmaddr, int shmflg)
```

Le premier argument est le numéro du descripteur retourné par la fonction shmget.

Voir http://linux.die.net/man/2/shmat

La fonction shmat

Le deuxième argument correspond à l'adresse de départ de la plage d'adresses relatives à attacher. Cette adresse peut être :

- NULL : on laisse le système déterminer la meilleure adresse (c'est le choix qui est utilisé la plupart du temps)
- une adresse (à déterminer pour des cas très spécifiques où la zone de mémoire partagée doit utiliser une plage d'adresses relatives précises, les adresses étant indiquées en dur dans le code par exemple)

Voir http://linux.die.net/man/2/shmat

La fonction shmat

Le troisième argument est un entier qui permet de contrôler le fonctionnement de shmat. Il est déterminé en faisant un ou logique entre 0 et les constantes ci-dessous :

- SHM_RDONLY : le segment de mémoire partagée est en lecture seule
- SHM_RND : l'adresse indiquée en deuxième argument est arrondie au multiple inférieur de SHMLBA (sinon cette adresse doit être alignée sur la limite d'une page mémoire).

Ce troisième argument est souvent mis à 0.

Voir http://linux.die.net/man/2/shmat

La fonction shmat

Si l'attachement se déroule correctement, la fonction renvoie l'adresse relative du premier octet sinon elle renvoie -1 et errno contient le code de l'erreur :

- EACCES : Les droits d'accès au segment de mémoire partagée ne sont pas bons ;
- EINVAL : le numéro de description n'est pas bon ou la valeur de l'adresse (deuxième argument) n'est pas bien aligné

Le segment de mémoire partagée se manipule donc simplement grâce à cette adresse (comme une zone de mémoire qui aurait été allouée par malloc).

Exemple

Voici un exemple de fonction qui alloue un segment de mémoire partagée et affecte une plage d'adresses relatives.

```
typedef struct SharedMem
 {
  int   descripteur;
  void* adresse;
 }
 MemoirePartagee;

MemoirePartagee superMalloc (int taille)
 {
  MemoirePartagee m;

  m.descripteur = shmget (IPC_PRIVATE, taille, IPC_CREAT|IPC_EXCL|0600);
  m.adresse     = (void*) -1; /* On suppose que ça se passe mal */

  if (m.descripteur != -1)
   {
    m.adresse = shmat (m.descripteur, NULL, 0);
   }

  return m;
 }
```

Le principe

Lorsque le segment de mémoire partagée n'est plus utile, il faut le détruire afin d'éviter les fuites de mémoire.

Cette destruction est d'autant plus importante que les segments de mémoire partagée survivent à la destruction des processus qui les a créés, car ils sont situés à l'extérieur des blocs mémoires affectés aux processus (contrairement à la mémoire réservée par l'instruction malloc qui se situe dans le tas, au sein du bloc de mémoire attribué au processus).

Le principe

Cette destruction s'effectue en deux temps :

- on doit d'abord détacher la plage d'adresses relatives afin que le système d'exploitation n'autorise plus les accès mémoires en utilisant ces adresses ;
- on désalloue ensuite le segment de mémoire afin de détruire l'objet IPC (libération de la mémoire, libération de la clé et libération du descripteur).

On constate que le processus de destruction se déroule dans le sens inverse de celui de la création.

La fonction shmdt

Elle est définie dans sys/shm.h et prend en paramètre l'adresse correspondant à la plage d'adresses attachée au segment.

Elle retourne 0 en cas de succès et -1 en cas d'échec (errno est initialisé à EINVAL signifiant que l'adresse passée en paramètre n'est pas bonne).

Le prototype de cette fonction est le suivant :

```
int shmdt(const void *shmaddr)
```

La fonction shmclt

Elle est définie dans sys/shm.h et prend en paramètre :

- le numéro de descripteur retourné par shmat
- une constante symbolique indiquant l'opération à réaliser (dans notre cas, il s'agit de IPC_RMID)
- un pointeur qui n'est pas significatif dans le cas d'une destruction (on le met à NULL dans ce cas)

Le prototype de cette fonction est le suivant :

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

Exemple

Voici un exemple de fonction qui libère la plage d'adresses relatives et désalloue le segment de mémoire partagée.

```
typedef struct SharedMem
 {
  int   descripteur;
  void* adresse;
 }
 MemoirePartagee;

int superFree (MemoirePartagee m)
 {
  int retour = shmdt (m.adresse);

  if (retour != -1)
   {
    retour = shmctl (m.descripteur, IPC_RMID, 0);
   }

  return retour;
 }
```

La fonction shmclt

La fonction shmclt peut effectuer deux autres actions en fonction du deuxième argument :

- IPC_STAT pour récupérer des informations concernant le segment de mémoire partagée
- IPC_SET qui permet de modifier les informations concernant ce même segment

Les informations récupérées ou les informations à modifier sont écrites dans un tampon dont l'adresse est indiquée dans le troisième argument de la fonction

La fonction shmclt

Le troisième argument correspond à un pointeur sur une structure shmid_ds définie dans sys/shm.h

```
struct shmid_ds
 {
  struct          ipc_perm shm_perm; /* Permissions d'accès       */
  int             shm_segsz;         /* Taille segment en octets  */
  time_t          shm_atime;         /* Heure dernier attachement */
  time_t          shm_dtime;         /* Heure dernier détachement */
  time_t          shm_ctime;         /* Heure dernier changement  */
  unsigned short  shm_cpid;          /* PID du créateur           */
  unsigned short  shm_lpid;          /* PID du dernier opérateur  */
  short           shm_nattch;        /* Nombre d'attachements     */
  /* ------------- Les champs suivants sont privés ------------- */
  unsigned short  shm_npages;        /* Taille segment en pages   */
  unsigned long   *shm_pages;        /* Taille d'une page (?)     */
  struct shm_desc *attaches;         /* Descript. attachements    */
 };
```

La fonction shmclt

Le dernier champ (attaches) est un pointeur sur une autre structure définie ci-dessous :

```
struct ipc_perm
 {
  key_t  key;
  ushort uid;   /* UID et GID effectifs du propriétaire   */
  ushort gid;
  ushort cuid;  /* UID et GID effectif du créateur        */
  ushort cgid;
  ushort mode;  /* Mode d'accès sur 9 bits de poids faible */
  ushort seq;   /* numéro de séquence                      */
 };
```

Exemple

On peut imaginer une fonction qui affiche des informations concernant un segment de mémoire partagée.

```
int afficher (MemoirePartagee m)
 {
  struct shmid_ds info;
  int retour = shmctl (m.descripteur, IPC_STAT, &info);

  if (retour!= -1)
   {
    printf ("Segment de memoire partagee de %d octets ", info.shm_segsz);
    printf ("cree par le processsus %d.\n", info.shm_cpid);
   }

  return retour;
 }
```

La fonction shmclt

Cette fonction renvoie 0 (succès) ou -1 (échec). Dans le second cas, errno peut être égal à :

- EACCES : on n'a pas les droits pour lire les informations concernant le segment de mémoire partagée
- EFAULT : le troisième argument n'est pas correct
- EINVAL : le premier ou le deuxième argument ne sont pas corrects
- EIDRM : le descripteur correspond à un segment de mémoire partagée qui a été détruit
- EPERM : on veut modifier ou supprimer un segment de mémoire partagée alors qu'on n'en est pas le propriétaire

Un programme complet

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

typedef struct SharedMem
 {
  int   descripteur;
  void* adresse;
 }
 MemoirePartagee;

MemoirePartagee superMalloc (int taille)
 {
  MemoirePartagee m;

  m.descripteur = shmget (IPC_PRIVATE, taille, IPC_CREAT|IPC_EXCL|0600);
  m.adresse     = (void*) -1; /* On suppose que ça se passe mal */

  if (m.descripteur != -1)
    m.adresse = shmat (m.descripteur, NULL, 0);

  return m;     }
```

Un programme complet

```
int superFree (MemoirePartagee m)
 {
  int retour = shmdt (m.adresse);

  if (retour != -1)
   {
    retour = shmctl (m.descripteur, IPC_RMID, 0);
   }

  return retour;
 }

int afficher (MemoirePartagee m)
 {
  struct shmid_ds info;
  int retour = shmctl (m.descripteur, IPC_STAT, &info);

  if (retour!= -1)
   {
    printf ("Segment de memoire partagee de %ld octets ", info.shm_segsz);
    printf ("cree par le processsus %d.\n", info.shm_cpid);
   }

  return retour;
 }
```

Un programme complet

```
int main (int argc, char** argv)
 {
  MemoirePartagee m = superMalloc (20);

  afficher (m);

  pid_t pid = fork ();
  pid_t pid2;
  int   etat;

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("Erreur");
              break;

    case  0 : printf ("Fils : PID=%d\n", getpid());  sleep (2); /* Attendre 2 secondes */
              printf ("Fils : Fin\n");               exit(0);

    default : printf ("Pere : PID=%d\n", getpid());
              pid2=wait(&etat);
              printf ("Pere : Le fils %d s'est arrete avec le code %d\n", pid2, etat);
              superFree (m);
              printf ("Pere : Fin\n");
   }

  return 0;
 }
```

Introduction

Les sémaphores IPC sont plus complexes que les sémaphores de Djkistra :

- ces objets permettent de regrouper plusieurs sémaphores. Il est donc possible de modifier plusieurs sémaphores en une seule instruction (semop).
- les instructions de modifications offrent plus de possibilités que les P et V de base (par exemple, l'équivalent de la primitive P n'est plus forcément bloquant)

La fonction semget

Cette fonction est définie dans sys/sem.h (il faut aussi inclure sys/types.h et sys/ipc.h) et a pour prototype :

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

Les trois paramètres de cette fonction sont les suivants :

- la clé qui peut être le joker IPC_PRIVATE ou une clé calculée grâce à la fonction ftok ;
- le nombre de sémaphores à placer dans cet objet IPC ;
- un entier permettant de spécifier les droits d'accès et les actions à réaliser (comme pour shmget).

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/semget.2.html

La fonction semget

Cette fonction renvoie un entier qui peut être :

- le numéro du descripteur de cet objet IPC en cas de succès ;
- -1 en cas d'erreur dont le code est stocké dans errno :
  - EACCES : le processus n'a pas le droit d'accéder au groupe de sémaphores désigné par key
  - EEXIST : le groupe de sémaphores existe déjà et les options IPC_CREAT et IPC_EXCL ont été activées
  - EIDRM : le groupe de sémaphores est sur le point d'être détruit

La fonction semget

- ENOENT : Aucun groupe de sémaphores associé a key n'existe et l'option IPC_CREAT n'est pas activée.
- ENOMEM : il n'y a pas assez de mémoire pour créer ces objets IPC
- ENOSPC : Le nombre de groupes de sémaphores dépasse le seuil SEMMNI, ou le nombre global de sémaphores a atteint son maximum (SEMMNS).

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/semget.2.html

La fonction semctl

Une fois que le groupe de sémaphores est effectivement créé, on va utiliser la fonction semctl (définie dans sys/sem.h) pour déterminer la valeur initiale des sémaphores.

Cette fonction, aussi utilisée à d'autres fins, a comme prototype :

```
int semctl (int semid, int semno, int cmd, union semun arg)
```

Le premier argument est le numéro de descripteur retourné par semget et le deuxième argument correspond au numéro du sémaphore dans le groupe.

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/semctl.2.html

La fonction semctl

Le troisième argument est un entier correspondant à l'action à accomplir (dans notre cas, il s'agit de SET_VAL).

Le dernier argument est de type « union semun », définie dans sys/sem.h de la manière suivante :

```
union semun
 {
  int              val;  /* pour SETVAL */
  struct semid_ds *buf;  /* pour IPC_STAT and IPC_SET */
  ushort          *array; /* pour GETALL and SETALL */
 };
```

Dans notre cas, on utilise le champ val.

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/semctl.2.html

Exemple

La fonction ci-dessous permet de créer un groupe, constitué d'un seul sémaphore, et d'en initialiser la valeur

```
int creerSemaphore (int _compteur)
 {
  int idSem = semget (IPC_PRIVATE, 1, 0666|IPC_CREAT|IPC_EXCL);

  semctl (idSem, 0, SETVAL, _compteur);

  return idSem;
 }
```

La fonction semctl

La fonction semctl est aussi utilisée pour détruire le groupe de sémaphores.

Pour cela, on met le numéro de descripteur comme premier argument et la constante IPC_RMID comme troisième argument (les deux autres arguments ne sont pas significatifs).

On peut avoir, par exemple, la fonction ci-dessous :

```
int detruireSemaphore (int _idSem)
 {
  return semctl (_idSem, 0, IPC_RMID, 0);
 }
```

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/semctl.2.html

La fonction semctl

La fonction semctl peut être utilisée à d'autres fins selon la valeur du troisième argument :

- IPC_STAT permet de récupérer les informations concernant l'objet IPC dans la structure semid_ds dont l'adresse est donnée dans le champ buf de l'union semun (le quatrième argument)
- IPC_SET est utilisée pour modifier certaines informations de l'objet IPC comme l'UID ou le GID (on utilise encore le champ buf de semun)
- GETALL permet de récupérer la valeur de tous les sémaphores du groupe afin de les placer dans le tableau pointé par array (le troisième champ de l'union semun)

La fonction semctl

- GETNCNT permet de connaître le nombre de processus bloqué sur le sémaphore dont le numéro est donné en deuxième argument, en attendant que le compteur de celui-ci augmente.
- GETPID est utilisé pour connaître le PID du processus qui a réalisé le dernier « semop » (en d'autres termes, un « P » ou un « V » sur le sémaphore dont le numéro est donné en deuxième argument
- GETVAL sert à connaître la valeur du compteur pour le sémaphore dont le numéro est donné en deuxième argument

Voir Voir http://manpagesfr.free.fr/man/man2/semop.2.html

La fonction semctl

- GETZCNT permet de connaître le nombre de processus bloqué sur le sémaphore dont le numéro est donné en deuxième argument, en attendant que le compteur de celui-ci passe à 0
- SETALL sert à initialiser les compteurs de tous les sémaphores en utilisant un tableau (champ array de la structure semun utilisée comme quatrième argument)
- SETVAL est utilisé pour le compteur du sémaphore, dont le numéro est fourni en deuxième argument, avec la valeur écrite dans le champ val de la structure semun du quatrième argument)

La fonction semop

La fonction semop permet d'effectuer des opérations sur les sémaphores (principalement faire évoluer la valeur du compteur).

Elle est définie dans sys/sem.h et a pour prototype :

```
int semop(int semid, struct sembuf *sops, unsigned nsops)
```

Le premier argument est le numéro de descripteur désignant le groupe de sémaphores.

Le deuxième argument est un pointeur sur une structure sembuf (un tableau d'éléments de type sembuf) et le dernier argument correspond à la taille de ce tableau.

La fonction semop

La structure sembuf est définie dans sys/sem.h de la manière suivante :

```
struct sembuf
 {
  unsigned short int sem_num;    /* # sémaphore */
  short              sem_op;     /* opération du sémaphore */
  short              sem_flg;    /* flags de l'opération */
 };
```

Le premier champ correspond au numéro, au sein du groupe, du sémaphore sur lequel on doit appliquer l'opération.

La fonction semop

Le deuxième champ correspond à l'opération elle-même :

- si sem_op est supérieur à 0, on ajoute cette valeur au compteur du sémaphore considéré
- si sem_op est nul, le processus se bloque jusqu'à ce que ce sémaphore voit son compteur devenir nul
- si sem_op est négatif, cette valeur va diminuer la valeur du compteur : si le compteur devient négatif, le processus est bloqué jusqu'à ce que le compteur devienne positif

La fonction semop

Le troisième champ permet de moduler le fonctionnement de semop sur cette opération, il s'agit d'un ou logique entre 0 et des constantes symboliques :

- IPC_NOWAIT rend semop non bloquant dans tous les cas de figure (si le compteur devient négatif, l'appel échoue)
- SEM_UNDO pour mettre à jour le compteur semadj qui cumule la valeur absolue de semop pour pouvoir annuler les effets des différentes opérations sur le sémaphore dès que le processus disparait

Voir http://manpagesfr.free.fr/man/man2/semop.2.html
http://souptonuts.sourceforge.net/code/IPC_README.html

La fonction semop

Si semop fonctionne correctement, elle renvoie 0, sinon elle renvoie -1 et errno vaut :

- E2BIG : la valeur de nsops dépasse SEMOPM (trop d'opérations à effectuer d'un coup)
- EACCES : le processus n'a pas les droits pour modifier les sémaphores
- EAGAIN : l'option IPC_NOWAIT a été activée alors que semop aurait du être bloquante (compteur qui devient négatif ou sem_op à 0)
- EFAULT : le pointeur sops n'est pas bon

La fonction semop

- EFBIG : la valeur dans le champ sem_num n'est pas bonne (inférieure à 0 ou supérieure ou égale au nombre de sémaphores dans le groupe).
- EIDRM : Le groupe de sémaphores a été supprimé.
- EINTR : un signal a été reçu pendant l'appel
- EINVAL : le groupe n'existe pas, le descripteur n'est bon ou la valeur de nsops n'est pas bonne
- ENOMEM : il n'y a pas assez de mémoire
- ERANGE : la somme de semop et du compteur d'un sémaphore est supérieur à SEMVMX.

Exemple

Les deux fonctions ci-dessous implémentent les primitives P et V de Dijsktra à l'aide des fonctions de la bibliothèque IPC

```
int P (int _idSemaphore)
 {
  struct sembuf sem;
  sem.sem_num = 0;
  sem.sem_op  = -1;
  sem.sem_flg = 0;
  return semop (_idSemaphore, &sem, 1);
 }

int V (int _idSemaphore)
 {
  struct sembuf sem;
  sem.sem_num = 0;
  sem.sem_op  = 1;
  sem.sem_flg = 0;
  return semop (_idSemaphore, &sem, 1);
 }
```

Le principe

Les files des messages sont un autre moyen de communiquer en complément des segments de mémoire commune.

Par rapport à ces dernières, les files (d'attente) de messages ordonnent les messages de manière à ce que le premier écrit soit le premier lu.

La fonction msgget

Cette fonction est définie dans sys/msg.h (il est aussi nécessaire d'inclure sys/types.h et sys/ipc.h) et a comme prototype :

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

Le premier argument est la clé qui désigne la file de messages (le joker IPC_PRIVATE ou une clé créée grâce à ftok).

Le second argument est utilisé pour spécifier les droits d'accès et les actions à réaliser (IPC_CREAT et IPC_EXCL)

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/msgget.2.html

La fonction msgget

Cette fonction renvoie :

- le numéro du descripteur de l'objet IPC si tout se passe bien
- -1 en cas d'échec, errno vaut alors :
  - EACCES : une file de messages, désignée par key, existe, mais on n'a pas la permission d'y accéder
  - EEXIST : une file de messages, désignée par key, existe et les options IPC_CREAT et IPC_EXCL sont activées
  - EIDRM : la file de messages est sur le point d'être supprimée.

La fonction msgget

- ENOENT : Il n'y a pas de file de messages désignée par key et l'option IPC_CREAT n'est pas activée.
- ENOMEM : Il n'y a pas assez de mémoire pour créer la file de messages.
- ENOSPC : Le nombre global de files de messages a atteint son maximum (MSGMNI).

La fonction msgctl

Cette destruction s'effectue grâce à la fonction msgctl, aussi définie dans sys/msg.h et dont le prototype est le suivant :

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

Le premier argument est la clé qui désigne la file de messages (IPC_PRIVATE ou une clé créée grâce à ftok).

Le deuxième argument est un entier correspondant à la commande à exécuter (dans notre cas IPC_RMID).

Le dernier argument est un pointeur sur une structure qui contient des informations complémentaires (étant sans signification ici, on le laisse à NULL).

La fonction msgsnd

L'envoi de messages s'effectue par la fonction msgsnd définie dans sys/msg.h de la manière suivante :

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

Le premier argument est la clé qui désigne la file de messages (IPC_PRIVATE ou une clé créée grâce à ftok).

Le deuxième argument est un pointeur sur une structure de données qui est utilisée comme tampon pour stocker les informations à envoyer.

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/msgsnd.2.html

La fonction msgsnd

Cette structure est à définir de la manière suivante :

```
struct msgbuf
 {
  long  mtype;          /* une étiquette >0 : type de message  */
  char  mtext [TAILLE]; /* contenu du message                  */
 };
```

Le troisième argument correspond au nombre d'octets à envoyer (souvent la taille du tableau mtext).

Le dernier argument permet de contrôler le fonctionnement de msgsnd. Le comportement par défaut de msgsnd est de bloquer le processus s'il n'y a pas assez de place pour envoyer le message. Si on active l'option IPC_NOWAIT, msgsnd devient non bloquant (mais échoue).

La fonction msgctl

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans le second cas, errno vaut :

- EACCES : on n'a pas le droit d'écrire dans la file de messages (et on n'a pas la capacité CAP_IPC_OWNER).
- EAGAIN : on n'a pas assez de places dans la file de messages pour envoyer les données et l'option IPC_NOWAIT est activée.
- EFAULT : l'adresse du tampon n'est pas bonne.
- EIDRM : La file de messages a été supprimée

La fonction msgctl

- EINTR : Sleeping on a full message queue condition, the process caught a signal.
- EINVAL : Invalid msqid value, or non-positive mtype value, or invalid msgsz value (less than 0 or greater than the system value MSGMAX).
- ENOMEM : The system does not have enough memory to make a copy of the message pointed to by msgp.

La fonction msgrcv

La récupération du message s'effectue, quant à elle, grâce à la fonction msgrcv, définie dans sys/msg.g comme il suit :

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
int msgflg);
```

Le premier argument est la clé qui désigne la file de messages.

Le deuxième argument est le pointeur sur la structure de données, de type msgbuf, qui va recueillir le message.

Le troisième argument indique le nombre d'octets qui doivent être lus depuis la file d'attente.

Voir http://linux.die.net/man/2/msgsnd

La fonction msgrcv

Le quatrième argument permet de spécifier les messages à lire depuis la file :

- Si msgtyp vaut 0, le premier message est lu.
- Si msgtyp est supérieur à 0 et que l'option MSG_EXCEPT n'est pas activée, alors on extrait le premier message où mtype est égal à msgtyp.
- Si msgtyp est supérieur à 0 et que l'option MSG_EXCEPT est activée, on extrait le premier message où mtype est différent de msgtyp.
- Si msgtyp est inférieur à 0, on extrait le premier message où mtype est inférieur ou égal à la valeur absolue de msgtyp.

La fonction msgrcv

Le dernier argument est un entier qui permet de contrôler le fonctionnement de msgrcv et qui est construit en faisant des ou logique entre 0 et les constantes symboliques suivantes :

- IPC_NOWAIT qui rend l'appel à msgrcv non bloquant même si aucun message n'est présent
- MSG_EXCEPT qui est utilisé lorsque msgtyp est supérieur à 0 pour lire des messages dont le champ msgtype est différent de msgtyp
- MSG_NOERROR qui tronque silencieusement les messages trop longs

La fonction msgrcv

En cas de succès, cette fonction renvoie le nombre d'octets lus depuis la file de messages. Dans le cas contraire, elle renvoie -1 et errno contient l'une des valeurs ci-dessous :

- E2BIG : la longueur du message est plus grande que la valeur de msgsz (troisième argument) et l'option MSG_NOERROR n'est pas activée
- EACCES : le processus n'a pas les droits sur la file de messages
- EAGAIN : aucun message n'est disponible et l'option IPC_NOWAIT est activée (rendant l'appel non bloquant)

La fonction msgrcv

- EFAULT : l'adresse spécifiée msgp (deuxième argument n'est pas bon)
- EIDRM : le message sur lequel le processus était en attente (donc bloqué sur l'appel de msgrcv) a été retiré
- EINTR : le processus qui était en attente d'un message a reçu un signal
- EINVAL : la valeur du premier argument (msgqid) ou du troisième (msgsz) n'est pas bon
- ENOMSG : aucun message répondant aux critères n'est présent et l'option IPC_NOWAIT est activée

La fonction msgctl

En plus de supprimer la file de messages, la fonction msgctl permet de :

- récupérer les informations concernant la file de messages (le deuxième argument vaut alors IPC_STAT)
- modifier les informations concernant la file de messages (le deuxième argument vaut alors IPC_SET)

Le troisième doit correspondre à un pointeur sur une structure msqid_ds qui va accueillir les informations recueillies ou qui va contenir les informations à modifier.

La fonction msgctl

Cette structure msqid_ds est définie dans sys/msg.h de la manière suivante :

```
struct msqid_ds
 {
  struct ipc_perm    msg_perm;    /* opération permission struct */
  struct __msg       *msg_first;  /* ptr to first message on q */
  struct __msg       *msg_last;   /* ptr to last message on q */
  unsigned short int msg_qnum;    /* # of messages on q */
  unsigned short int msg_qbytes;  /* max # of bytes on q */
  pid_t              msg_lspid;   /* pid of last msgsnd */
  pid_t              msg_lrpid;   /* pid of last msgrcv */
  time_t             msg_stime;   /* last msgsnd time */
  time_t             msg_rtime;   /* last msgrcv time */
  time_t             msg_ctime;   /* last change time */
  unsigned short int msg_cbytes;  /* current # bytes on q */
  char               msg_pad[22]; /* room for future expansion */
 };
```

La fonction msgctl

Cette fonction renvoie 0 succès) ou -1 (échec). La nature de ce problème est codée dans la variable errno :

- EACCES : le processus n'a pas les droits pour récupérer les informations stockées dans la structure msqid
- EFAULT : le troisième argument (le pointeur sur la structure msqid_ds) n'est pas bon
- EINVAL : la valeur du premier ou du deuxième argument n'est pas bon
- EPERM : l'UID effectif du processus appelant n'a pas les droits pour réaliser une opération IPC_SET ou IPC_RMID
- EIDRM : La file de messages a été supprimée.

Exemple d'utilisation des files de messages IPC

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define TAILLE 10

typedef struct
 {
  long mtype;             /* type de message (etiquette) > 0 */
  char mtext[TAILLE];     /* contenu du message              */
 }
 msgbuf;

int main (int argc, char** argv)
 {
  int    fileDeMessages = msgget (IPC_PRIVATE, 0666);
  msgbuf message;

  pid_t pid = fork ();

  pid_t pid2;
  int   etat;
```

Exemple d'utilisation des files de messages IPC

```
switch (pid)
 {
  case -1 : printf ("Erreur dans la creation du processus fils.\n");
            perror ("Erreur");
            break;

  case  0 : {
             char messageRecu [TAILLE];
             int  tailleMessageRecu = 0;

             printf ("Fils (PID=%d) ", getpid());

             /* On initialise messageRecu avec des 0 */
             bzero (messageRecu, TAILLE);

             sleep (2); /* Attendre 2 secondes */

             /* On initialise le tableau mtext avec des 0
              * On écrit dans la case 0 de mtext, la taille du texte à envoyer
              * On écrit 1 dans mtype (un choix arbitraire, les messages de
              * type 1 correspondent à une taille) */

             bzero (message.mtext, TAILLE);
             msgrcv (fileDeMessages, &message, 1, 1, 0);

             tailleMessageRecu = message.mtext[0];

             printf ("a recu %d octets : ", tailleMessageRecu);
```

Exemple d'utilisation des files de messages IPC

```
            /* On initialise le tableau mtext avec des 0
             * On écrit dans la case 0 de mtext, la taille du texte à envoyer
             * On écrit 1 dans mtype (un choix arbitraire, les messages de
             * type 1 correspondent à une taille) */

            bzero (message.mtext, TAILLE);
            msgrcv (fileDeMessages, &message, tailleMessageRecu, 2, 0);
            strcpy (messageRecu, message.mtext);

            printf ("%s\n", messageRecu);

            printf ("Fils : Fin\n");
            exit(0);
           }

default : {
            char* messageAEnvoyer = "Bonjour";
            printf ("Pere (PID=%d) ", getpid());

            /* On initialise le tableau mtext avec des 0
             * On écrit dans la case 0 de mtext, la taille du texte à envoyer
             * On écrit 1 dans mtype (un choix arbitraire, les messages de
             * type 1 correspondent à une taille) */

            bzero (message.mtext, TAILLE);
            message.mtext[0] = strlen (messageAEnvoyer);
            message.mtype    = 1;
            msgsnd (fileDeMessages, &message, 1, 0);
```

Exemple d'utilisation des files de messages IPC

```
                printf ("%d octets : ", message.mtext[0]);

                /* On initialise de nouveau le tab6eau mtext avec des 0
                 * On copy la chaine à envoyer dans mtext
                 * On écrit 2 dans mtype (un choix arbitraire, les messages de
                 * type 2 correspondent à un texte) */

                bzero (message.mtext, TAILLE);
                strcpy (message.mtext, messageAEnvoyer);
                message.mtype = 2;
                msgsnd (fileDeMessages, &message, strlen (messageAEnvoyer), 0);

                printf ("%s\n", messageAEnvoyer);

                pid2=wait(&etat);
                printf ("Pere : Le fils %d s'est arrete avec le code %d\n", pid2, etat);
                msgctl (fileDeMessages, IPC_RMID, NULL);
                printf ("Pere : Fin\n");
               }
  }

 return 0;
}
```

La commande ipcs

Depuis le shell, la commande ipcs liste les objets IPC présents en mémoire. Elle accepte les options suivantes :

- -i id pour afficher l'objet IPC ayant cet identifiant
- -m pour n'afficher que les segments de mémoire
- -q pour n'afficher que les files de messages
- -s pour n'afficher que les sémaphores
- -a pour tout afficher (choix par défaut)
- -t pour afficher les dates de création, modification...
- -p pour afficher les processus créateur et utilisateur
- -c pour afficher les noms des propriétaires des processus
- -l pour afficher les nombres maximums de sémaphores...
- -u pour afficher un résumé des objets IPC existants

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man8/ipcs.8.html

La commande ipcrm

La commande ipcrm permet de supprimer les objets IPC de la mémoire. Elle accepte les options suivantes :

- « -M clé » ou « -m id » pour supprimer un segment de mémoire partagée grâce à sa clé ou son identifiant ;
- « -S clé » ou « -s id » pour supprimer un sémaphore grâce à sa clé ou son identifiant ;
- « -Q clé » ou « -q id » pour supprimer une file de messages grâce à sa clé ou son identifiant

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man8/ipcrm.8.html

La commande ipcmk

La commande ipcmk permet de créer des objets IPC. Elle accepte les options suivantes :

- « -M taille » pour créer un segment de mémoire partagée dont la taille en octet est donnée en paramètre ;
- « -S nbSem » pour créer un groupe de nbSem sémaphores ;
- « -Q » pour créer une file de messages ;
- « -p mode » permet de spécifier les droits d'accès (par défaut 644)

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man8/ipcrm.8.html

Les signaux

Le principe

Un signal est une **notification** associée à un **événement** engendré par la partie matérielle ou logicielle :

- Certains signaux peuvent être lancés à partir d'un **terminal** grâce aux caractères spéciaux comme intr, quit dont la frappe est transformée en l'envoi des signaux SIGINT et SIGQUIT ;
- D'autres sont dus à des **causes internes au processus**, par exemple : SIGSEGV est envoyée en cas d'erreur d'adressage, SIGFPE signale une division par zéro (Floating Point Exception).

Les 32 signaux

Il existe 32 signaux qui peuvent être manipulés grâce à des constantes symboliques définies dans signal.h.

	Identificateur	Signification
1	SIGHUP	Instruction (HANG UP) - Fin de session
2	SIGINT	Interruption
3	SIGQUIT	Interruption
4	SIGILL	Instruction illégale
5	SIGTRAP	Trace trap
6	SIGABRT	(ANSI) Instruction (ABORT)
	SIGIOT	(BSD) IOT Trap
7	SIGBUS	Bus error
8	SIGFPE	Floating-point exception - Exception arithmétique
9	SIGKILL	Instruction (KILL) - termine le processus immédiatement

Les 32 signaux

	Identificateur	Signification
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Violation de mémoire
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Broken PIPE - Erreur PIPE sans lecteur
14	SIGALRM	Alarme horloge
15	SIGTERM	Signal de terminaison
16	SIGSTKFLT	Stack Fault
17	SIGCHLD	Modification du statut d'un processus fils
18	SIGCONT	Demande de reprise du processus
19	SIGSTOP	Demande de suspension imbloquable
20	SIGTSTP	Demande de suspension depuis le clavier
21	SIGTTIN	Lecture terminal en arrière-plan

Les 32 signaux

	Identificateur	Signification
22	SIGTTOU	Ecriture terminal en arrière-plan
23	SIGURG	Évènement urgent sur socket
24	SIGXCPU	Signal utilisateur 2
25	SIGXFSZ	Broken PIPE - Erreur PIPE sans lecteur
26	SIGVTALRM	Alarme horloge
27	SIGPROF	Signal de terminaison
28	SIGWINCH	Stack Fault
29	SIGPOLL	(System V) occurrence d'un évènement attendu
	SIGIO	(BSD) I/O possible actuellement
30	SIGPWR	Power failure restart
31	SIGSYS	Erreur d'appel système
32	SIGUNUSED	

La fonction kill

On peut envoyer des signaux depuis un processus grâce à la fonction **kill** définie dans **signal.h** :

```
int kill(pid_t pid, int sig)
```

L'argument pid peut prendre 4 valeurs :

- une valeur strictement positive qui désigne un processus
- 0 pour envoyer le signal à tous les processus du groupe auquel appartient le processus courant
- -1 pour atteindre tous les processus de l'utilisateur
- une valeur strictement inférieure à -1 pour envoyer le signal au groupe de processus dont le GID est la valeur absolue de l'argument pid

Voir http://linux.die.net/man/2/kill

La fonction kill

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans le second cas, errno peut valoir :

- EINVAL : la valeur de sig est incorrecte
- EPERM : le processus n'a pas la permission d'envoyer un signal au(x) processus cible(s)
- ESRCH : le PID ou le GID n'existe pas (cela peut aussi se produire dans le cas d'un processus zombie)

Voir http://linux.die.net/man/2/kill

La fonction kill

Le processus père crée un fils puis il lui envoie le signal SIGTERM afin de lui demander de s'arrêter.

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

int main (int argc, char** argv)
 {
  pid_t pid = fork ();
  pid_t pid2;
  int   etat;

  switch (pid)
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("Erreur");
              break;
    case  0 : printf ("On se trouve dans le processus fils.\n"); while (1); break;
    default : printf ("On se trouve dans le processus pere.\n"); sleep (2);
              kill (pid, SIGTERM);
              pid2 = wait(&etat);
              printf ("Mort du fils de pid %d avec etat=%d\n",pid2, etat);
   }
  return 0;  }
```

La fonction killpg

La fonction killpg (aussi définie dans signal.h) est un raccourci qui permet d'envoyer un signal à l'ensemble des processus d'un groupe. Sa signature est la suivante :

```
int killpg(int gid, int sig)
```

gid peut avoir une valeur correspondant à un groupe particulier ou être mis à 0. Dans ce second cas, killpg envoie le signal au groupe dans lequel se trouve processus appelant.

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec.

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/killpg.2.html

La fonction killpg

Lors de l'exécution de killpg, les erreurs suivantes peuvent survenir :

- EINVAL : la valeur de sig n'est pas valide
- ESRCH : le GID est non valide
- EPERM : le processus appelant n'appartient pas à root et il y a au moins un processus du groupe cible qui n'appartient pas à l'utilisateur du processus appelant (l'UID effectif n'est pas bon)

La commande kill

L'envoi d'un signal peut aussi s'effectuer depuis le shell en utilisant la commande **kill**. Le premier paramètre est le numéro du signal et le second le PID (ou le GID) :

- « kill -15 124 » signifie qu'on demande au processus n°124 de s'arrêter (on lui laisse le temps d'effectuer les dernières écritures...)
- « kill -9 236 » signifie que le processus n°236 doit s'arrêter immédiatement
- « kill -9 -1 » provoque la fermeture de tous les processus d'un utilisateur (à éviter quand on est root)

Voir http://linux.die.net/man/1/kill

La fonction alarm

Cette fonction permet un envoi différé du signal SIGALRM au processus appelant. Elle est définie dans unistd.h et sa signature est la suivante :

```
unsigned int alarm(unsigned int nb_sec)
```

Elle prend en paramètre le nombre de secondes qui doit s'écouler entre l'appel de la fonction et l'envoi du signal.

Les appels successifs annulent les précédents. Cette fonction renvoie soit le nombre de secondes qu'il restait avant l'envoi de SIGALRM, consécutif à un appel précédent, soit 0 (aucun appel n'ayant été effectué auparavant).

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/alarm.2.html

Le principe

Lorsqu'un processus reçoit un signal, il accomplit une action qui lui est associée par défaut. Par exemple, si le processus reçoit le signal SIGTERM, il arrête son fonctionnement.

Il est possible d'associer d'autres traitements (implémentés au sein de fonctions) à ces signaux afin de les adapter à nos besoins.

Il existe deux manières de procéder :

- une méthode « ancienne » associé à Unix
- une méthode plus récente, certifiée Posix, donc ayant une meilleure portabilité.

Le gestionnaire de signal

On implante le comportement souhaité dans une fonction qui doit obligatoirement prendre en paramètre en int et avoir comme type de retour void.

L'argument de type int permettra de passer le numéro du signal en paramètre comme le montre l'exemple ci-dessous.

```
void capter (int n)
 {
  printf(" J'ai capté le signal %d\n",n);
 }
```

La fonction signal

On utilise ensuite la fonction signal définie dans signal.h et dont la signature est la suivante :

```
sighandler_t signal(int           signum,
                    sighandler_t handler)
```

sighandler_t est un type définie dans signal.h qui correspond à un pointeur sur une fonction prenant en paramètre un int et retournant un void

```
typedef void (*sighandler_t)(int)
```

Voir http://linux.die.net/man/2/signal
http://fr.wikibooks.org/wiki/Programmation_C/Gestion_des_signaux
http://drocourt.info/cours/Unix/C-Programmation_Systeme/prog_sys4.xhtml
http://www.gnu.org/s/hello/manual/libc/Signal-Actions.html

La fonction signal

Cette fonction prend, comme premier argument, le numéro du signal concerné (ou la constante symbolique).

Le second argument peut être :

- SIG_DFL : pour indiquer que le signal doit être traité en utilisant le comportement par défaut
- SIG_IGN : pour indiquer que le signal doit être ignoré
- Le pointeur sur la fonction (autrement dit le nom de la fonction) chargée de traiter le signal.

Remarque : les signaux SIGKILL and SIGSTOP ne peuvent être ni ignorés, ni interceptés

La fonction signal

La fonction signal renvoie une variable de type sighandler_t qui peut être

- en cas de succès, l'ancienne méthode de gestion (SIG_DFL, SIG_IGN ou un pointeur de fonction)
- en cas d'échec, SIG_ERR.

Dans le second cas, errno est mis à EINVAL signifiant que la valeur du signal est non valide ou égal à SIGKILL ou SIGSTOP.

Un premier exemple d'utilisation

L'exemple ci-dessous montre comment le signal SIGINT (les touches [CTRL] et [C]). Un message d'alerte est affiché, mais le programme ne s'arrête pas.

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

void capter (int n)
 {
  printf("J'ai capté le signal SIGINT (%d)\n",n);
 }

int main (int argc, char** argv)
 {
  signal (SIGINT, capter);

  while (1);
  return 0;
 }
```

Un premier exemple d'utilisation

L'exemple précédent est incomplet sur certains systèmes d'exploitation. En effet, si on envoie deux fois de suite le signal SIGTERM, le programme s'arrête.

En effet, dès que le signal est intercepté et traité par la fonction capter, le gestionnaire par défaut est automatiquement remis en place. Pour contourner ce problème, il faut donc ajouter l'instruction signal au début de la fonction de traitement.

```
void capter (int n)
 {
  signal (SIGINT, capter);
  printf("J'ai capté le signal SIGINT (%d)\n",n);
 }
```

L'attente d'un signal

La fonction pause définie dans unistd.h force le processus à attendre l'arrivée d'un signal. Le prototype de cette fonction est la suivante :

```
int pause(void)
```

Comme la fonction pause ne se termine que si un signal a été intercepté, elle retournera toujours -1 et errno sera initialisé à EINTR signifiant qu'un un signal a été intercepté et que le gestionnaire a fini de le traiter.

Voir http://linux.die.net/man/2/pause

La structure sigaction

Le comportement de la fonction signal peut varier d'un Unix à l'autre, il est donc conseillé d'utiliser cette seconde méthode, car elle fait partie de la norme **Posix**.

Le traitement des signaux s'effectue, cette fois-ci, en utilisant la structure ci-dessous définie dans signal.h.

```
struct sigaction
 {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);  /*obsolete : a ne pas utiliser */
 }
```

La structure sigaction

Le premier champ (sa_handler) peut stocker :

- un pointeur sur la fonction chargée de traiter le signal ;
- la macro SIG_DFL correspondant au traitement par défaut ;
- la macro SIG_IGN utilisée lorsque le signal doit être ignoré.

Le deuxième champ (sa_sigaction) sera décrit un peu plus loin. Il est apparu avec le noyau Linux 2.1.86 et joue un rôle similaire à sa_handler (la différence est qu'on peut manipuler des fonctions de gestion qui prennent 3 arguments au lieu d'un simple int).

Voir http://www.linux-france.org/article/man-fr/man2/sigaction-2.html http://manpagesfr.free.fr/

La structure sigaction

Le troisième champ (sa_mask) est un masque qui permet de spécifier les signaux qui doivent être bloqués durant l'exécution de la fonction, pointée par sa_handler, ou de la macro SIG_DFL (cela permet d'éviter qu'un signal perturbe le traitement d'un autre signal).

Le quatrième champ (sa_flag) permet de contrôler le fonctionnement de sigaction. Il s'agit d'un entier dont la valeur est un ou logique entre 0 et des constantes présentées dans les pages suivantes.

Voir http://www.labri.fr/perso/billaud/travaux/SYSRESEAU/HTML/sysreseau-13.html
http://pubs.opengroup.org/onlinepubs/009695399/functions/sigaction.html
http://www.linux-france.org/article/man-fr/man2/sigaction-2.html
http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/sigaction.2.html
http://shtroumbiniouf.free.fr/CoursInfo/Systeme2/TP/FctSystUnix/Signaux.html

La structure sigaction

- SA_NOMASK ou SA_NODEFER :

  Lorsqu'un signal est capté, il est automatiquement ajouté au masque afin de ne pas être capté de nouveau durant le fonctionnement de la fonction de traitement. Utiliser cette option permet donc à un signal d'être capté durant le fonctionnement de son gestionnaire.

- SA_ONESHOT or SA_RESETHAND :

  Lorsque le signal a été traité, la structure sigaction est modifiée de manière à appeler le gestionnaire par défaut lors du prochain appel (cela permet d'avoir un comportement comparable à la fonction signal).

La structure sigaction

- SA_RESTART :

  Certaines fonctions bloquantes comme wait, read, accept (socket) … peuvent échouer si une interruption survient*. L'option SA_RESTART relance automatiquement ces fonctions

- SA_NOCLDSTOP :

  Si on reçoit un signal SIGCHLD, on ne doit pas le considérer s'il est consécutif à la réception, par le processus fils, des signaux SIGSTOP, SIGTSTP, SIGTTIN, ou SIGTTOU.

*Voir par exemple http://manpagesfr.free.fr/man/man2/wait.2.html

Les fonctions de gestion du masque

Lorsque la structure est construite, on utilise les fonctions ci-dessous pour modifier le masque (le champ sa_mask de type sigset_t) :

- int sigemptyset (sigset_t *set) permet d'initialiser le masque ;
- int sigfillset (sigset_t *set) remplit le masque à tous les signaux ;
- int sigaddset (sigset_t *set, int signum) ajoute un signal particulier au masque ;
- int sigdelset (sigset_t *set, int signum) retire du masque le signal dont le numéro est passé en paramètre

Les fonctions de gestion du masque

- int sigismember (const sigset_t *set, int signum) teste si un signal appartient au masque ;
- La fonction int sigprocmask(int how, const sigset_t *set, sigset_t *oldset) permet de combiner deux masques (set et oldset) selon la méthode spécifiée dans le champ how :
  - SIG_SETMASK : affectation d'un nouvel ensemble
  - SIG_BLOCK : union des deux ensembles
  - SIG_UNBLOCK : ancien ensemble – nouvel ensemble

Les fonctions de gestion du masque

- int sigpending (sigset_t *set) retourne l'ensemble des signaux bloqués (reçus, mais en attente d'être délivrés) dans la structure pointée par set

  Si tout se déroule correctement, la fonction renvoie 0, sinon elle renvoie -1 et errno vaut EFAULT.

La fonction sigaction

Il ne reste plus qu'à utiliser la fonction sigaction pour affecter la structure sigaction à un signal. Sa signature est la suivante :

```
int sigaction (      int                 signal,
               const struct sigaction *actionCourante
                     struct sigaction *actionPrecedente)
```

Cette fonction utilise :

- un champ signal qui correspond au numéro du signal à intercepter ;
- un pointeur sur la structure sigaction afin de lier le signal à sa nouvelle méthode de gestion.

La fonction sigaction

Le troisième champ (actionPrecedente) est aussi un pointeur sur la structure sigaction qui permet de récupérer l'ancienne méthode de gestion*.

Cette fonction renvoie 0 en cas de succès et -1 sinon. errno peut alors contenir l'une des valeurs suivantes :

- EINVAL : l'argument signal n'est pas correct (on essaye, par exemple, d'intercepter SIGKILL ou SIGSTOP) ;
- EFAULT : les valeurs des arguments actionPrecedente ou actionCourante ne sont pas correctes ;
- EINTR : l'exécution de sigaction a été interrompue.

Exemple

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

void handler (int signalRecu)
 {
  printf("Je receptionne le signal %d\n",signalRecu); fflush(stdout);
 }

int main (int argc, char** argv)
 {
  struct sigaction prepaSignal;
                   prepaSignal.sa_handler=&handler;
                   prepaSignal.sa_flags=0;

  sigemptyset(&prepaSignal.sa_mask);

  sigaction(SIGINT, &prepaSignal, NULL);
  sigaction(SIGQUIT,&prepaSignal, NULL);
  sigaction(SIGTERM,&prepaSignal, NULL);

  while (1);
  return 0;
 }
```

Un gestionnaire plus complet

Par défaut, le gestion ne prend en paramètre qu'un entier correspondant au numéro du signal. Depuis le noyau Linux 2.1.86, il est possible de créer des gestionnaires respectant la signature ci-dessous :

```
void (*sa_sigaction)(int, siginfo_t *, void *)
```

Le premier argument est l'entier qui correspond toujours au numéro du signal intercepté (renseigné automatiquement lors de l'appel de la fonction).

Un gestionnaire plus complet

Le deuxième argument est un pointeur sur une structure siginfo_t qui comporte les champs suivants :

```
int      si_signo;      /* Numéro de signal          */
int      si_errno;      /* Numéro d'erreur           */
int      si_code;       /* Code du signal            */
pid_t    si_pid;        /* PID de l'émetteur         */
uid_t    si_uid;        /* UID réel de l'émetteur    */
int      si_status;     /* Valeur de sortie          */
clock_t  si_utime;      /* Temps utilisateur écoulé  */
clock_t  si_stime;      /* Temps système écoulé      */
sigval_t si_value;      /* Valeur de signal          */
int      si_int;        /* Signal Posix.1b           */
void *   si_ptr;        /* Signal Posix.1b           */
void *   si_addr;       /* Emplacement d'erreur      */
int      si_band;       /* Band event                */
int      si_fd;         /* Descripteur de fichier    */
```

Un gestionnaire plus complet

Grâce à ces différents champs, nous pouvons déterminer, par exemple, le PID de l'émetteur du signal ou l'identifiant de l'utilisateur UID (ce qui n'était pas possible avec la première méthode ou le premier type de gestionnaire).

Enfin le dernier argument est un pointeur sur void qui désigne en réalité le contexte d'exécution, stocké dans une structure ucontext (définie dans sys/ucontext.h).

La manipulation des contextes d'exécution étant hors de propos dans ce cours, ils ne seront donc pas abordés.

Voir http://books.google.com/books?id=VVUbTZnknFIC&pg=PT338#v=onepage&q&f=false

Exemple

L'exemple de la page suivante est la traduction de l'exemple précédent.

On remarque les modifications opérées au niveau du code du gestionnaire de l'interruption, mais également dans le main où :

- on a utilisé le champ sa_sigaction (au lieu du champ sa_handler) ;
- on a activé l'option SA_SIGINFO au niveau du champ sa_flags

Exemple

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

void fct (int signalRecu, siginfo_t* info, void* pasUtileIci)
 {
  printf("Je receptionne le signal %d envoye par le processus %d\n",
         signalRecu, info->si_pid); fflush(stdout);
 }

int main (int argc, char** argv)
 {
  struct sigaction prepaSignal;
                   prepaSignal.sa_sigaction=&fct;
                   prepaSignal.sa_flags=SA_SIGINFO;

  sigemptyset(&prepaSignal.sa_mask);
  sigaction(SIGINT, &prepaSignal, NULL);
  sigaction(SIGQUIT,&prepaSignal, NULL);
  sigaction(SIGTERM,&prepaSignal, NULL);

  while (1);
  return 0;  }
```

L'attente d'un signal

Pour forcer un processus à attendre l'arrivée d'un signal, on peut utiliser la fonction sigsuspend dont le prototype est :

```
int sigsuspend(const sigset_t *mask)
```

Cette fonction remplace temporairement le masque spécifié dans le champ sa_mask. Elle bloque alors le processus jusqu'à l'arrivée d'un signal qui ne soit pas bloqué. Lors du déblocage, le masque est remis à son état antérieur.

Cette fonction renvoie 0 en cas de succès, -1 sinon. Dans le second cas errno est mis à EFAULT (la valeur de mask est incorrecte) ou EINTR (l'appel est interrompu par un signal).

Voir http://books.google.com/books?id=VVUbTZnknFIC&pg=PT338#v=onepage&q&f=false

Les tubes

Le principe

Les tubes sont un mécanisme de communication lié au système de gestion de fichiers. Ces tubes nommés ou anonymes correspondent à des paires d'entrées dans la table des fichiers :

- l'entrée 0 pour la lecture du tube ;
- l'entrée 1 pour l'écriture dans le tube.

Les tubes ont un fonctionnement FIFO : le premier élément entré est le premier sorti

La sortie (la lecture) d'un élément est destructive

La fonction pipe

Pour créer un tube anonyme, on utilise l'instruction pipe définie dans unistd.h. Sa signature est la suivante :

```
int pipe (int filedes[2])
```

Elle prend en paramètre un tableau de 2 entiers :

- la première case contient le descripteur de fichier pour la lecture
- la seconde case stocke le descripteur de fichier pour l'écriture

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec

La fonction pipe

Un processus ne peut utiliser que les tubes qu'il a créés ou hérités. Ces tubes sont détruits lorsque les processus qui les utilisent sont détruits.

En général, le père créé un tube puis fait un fork : le tube devient alors une ressource commune grâce à laquelle les deux processus peuvent alors échanger.

Par mesure de sécurité, le père et le fils vont fermer l'extrémité du tube qu'ils n'utilisent pas (grâce à la fonction close avec le descripteur de fichier non utilisé en paramètre)

La fonction pipe

En cas d'échec, errno peut contenir l'une des valeurs suivantes :

- EFAULT : le tableau d'entier est non valide
- EMFILE : le processus utilise trop de descripteurs de fichier
- ENFILE : le nombre global de fichiers ouverts est à son maximum

Voir http://linux.die.net/man/2/pipe

Exemple

```
#include <stdio.h>
#include <unistd.h>
int tube[2];
char buf[20];

int main(int argc, char** argv)
 {
  pipe(tube);

  switch (fork())
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("Erreur");
              break;

    case  0 : printf ("Fils : PID=%d\n", getpid());
              close(tube[0]);  // On ferme le côté lecture
              /* Suite du programme */
              break;

    default : printf ("Pere : PID=%d\n", getpid());
              close(tube[1]);  // On ferme le côté écriture
              /* Suite du programme */
   }
  return 0;
 }
```

La fonction read

Elle s'effectue par l'intermédiaire de la fonction read définie dans unistd.h de la manière suivante :

```
ssize_t read(int desc, void *tampon, size_t nbOctets)
```

Les arguments sont les suivants :

- le descripteur de fichier utilisé pour la lecture ;
- un pointeur vers un tampon qui va recueillir les données ;
- le nombre d'octets à récupérer.

Cette fonction renvoie le nombre d'octets lus en cas de succès et -1 en cas d'échec.

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/read.2.html

La fonction read

En cas d'erreur, errno peut avoir comme valeur :

- EINTR : la fonction read a été interrompue par l'arrivée d'un signal
- EAGAIN : la lecture est non bloquante et aucune donnée n'était disponible.
- EIO : une erreur d'entrée/sortie est survenue
- EBADF : le descripteur de fichier n'est pas valide
- EINVAL : le descripteur de fichier ne correspond à un objet permettant la lecture
- EFAULT : l'adresse du tampon n'est pas bonne.

L'algorithme de lecture

La lecture s'effectue selon l'algorithme ci-dessous :

```
si le tube n'est pas vide
   alors lecture de min(nb octets présents, nb octets demandés)
   sinon
      si le nombre d'écrivains est nul
         alors c'est la fin de fichier et la fonction renvoie 0
         sinon
            si lecture bloquante
               alors le processus est mis en sommeil
               sinon en fonction de l'indicateur
                  si indicateur = O_NONBLOCK
                     alors la fonction retourne -1 et errno=EAGAIN
                  fsi
                  si indicateur = O_NDELAY
                     alors la fonction retourne 0
                  fsi
            fsi
      fsi
fsi
```

La fonction write

Elle est réalisée grâce à la fonction write définie dans unistd.h de la manière suivante :

```
ssize_t write(int desc, const void *tampon, size_t nbOctets)
```

Le premier paramètre est le descripteur de fichier utilisé pour l'écriture

Le deuxième est le pointeur vers le tampon qui contient les données à envoyer

Le dernier est le nombre d'octets à envoyer (qui doit être inférieur à PIPE_BUF défini dans <limits.h>)

Voir http://www.linux-kheops.com/doc/man/manfr/man-html-0.9/man2/write.2.html

La fonction write

En cas d'erreur, errno peut avoir comme valeur :

- EINTR : la fonction read a été interrompue par l'arrivée d'un signal
- EAGAIN : la lecture est non bloquante et aucune donnée n'était disponible.
- EIO : une erreur d'entrée/sortie est survenue
- EBADF : le descripteur de fichier n'est pas valide
- EINVAL : le descripteur de fichier ne correspond à un objet permettant la lecture
- EFAULT : l'adresse du tampon n'est pas bonne.

L'algorithme d'écriture

La lecture s'effectue selon l'algorithme ci-dessous :

```
si le nombre de lecteurs est nul
   alors envoi du signal SIGPIPE à l'écrivain.
   sinon
      si l'écriture est bloquante
         alors la fonction rend la main que lorsque les n caractères
               ont été écrits dans le tube.
         sinon
            si n > PIPE BUF
               alors la fonction renvoie un nombre inférieur à n
                     (éventuellement -1)

               sinon
                  si assez d'emplacements libres
                     alors écriture de n caractères et renvoi de n
                     sinon aucun caractère écrit et renvoi de 0 ou -1
                  fsi
            fsi
      fsi
fsi
```

La fonction fcntl

Pour modifier l'accès à un fichier afin de le rendre bloquant ou non, on utilise la fonction fctnl définie dans unistd.h

```
int fcntl(int fd, int cmd, long arg)
```

Le premier argument correspond au descripteur de fichier

Le second argument est une constante symbolique correspondant à une commande à appliquer au fichier :

- F_GETFD permet de récupérer les différents drapeaux correspondant au fichier désigné par le descripteur fd
- F_SETFD est utilisé pour modifier des drapeaux afin de contrôler l'accès au fichier (ces drapeaux sont contenus dans le 3ème argument de la fonction)

La fonction fcntl

Les drapeaux qui peuvent être utilisés sont les suivants :

- O_APPEND : ajouter les nouvelles écritures à la fin du fichier
- O_ASYNC : envoyer le signal SIGIO dès que des nouvelles données sont disponibles dans le tube
- O_DIRECT : écrire directement dans le fichier sans passer par le cache du système
- O_NOATIME : ne pas horodater le dernier accès au fichier
- O_NONBLOCK : rendre les accès non bloquants

Voir http://manpages.ubuntu.com/manpages/maverick/fr/man7/pipe.7.html

Exemple d'une fonction de blocage/déblocage

Pour écrire cette fonction, on droit récupérer la valeur courante des différents drapeaux sous forme d'un entier puis ne modifier qu'un seul bit :

- On fait un OU logique entre l'entier contenant les drapeaux (flags) et la constante O_NONBLOCK (où un seul bit est à 1) pour rendre les accès non bloquants :

```
flags | O_NONBLOCK
```

- on fait un ET logique entre ce même entier flags et l'inverse (NON logique) de O_NONBLOCK (on a un seul bit à 0) pour rendre les accès bloquants

```
flags & ~O_NONBLOCK
```

Exemple d'une fonction de blocage/déblocage

On obtient le code ci-dessous

```
int SetBlockMode (int desc, bool blocking)
 {
  int flags = fcntl (desc, F_GETFL);
  int r;

  if (blocking == true)
        r = fcntl (desc, F_SETFL, flags & ~O_NONBLOCK);
   else r = fcntl (desc, F_SETFL, flags |  O_NONBLOCK);

  return r;
 }
```

Exemple 1

Un premier exemple où le fils envoie le message « bonjour » au père.

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

int tube[2];      char buf[20];

int main(int argc, char** argv)
 {
  pipe(tube);

  switch (fork())
   {
    case -1 : printf ("Erreur dans la creation du processus fils.\n"); perror ("Erreur");
              break;

    case  0 : printf ("Fils : PID=%d\n", getpid());    close(tube[0]);
              write(tube[1], "\"bonjour\"", 9); exit(0);

    default : printf ("Pere : PID=%d\n", getpid());    close(tube[1]);    wait(NULL);
              read(tube[0], buf, 9); printf("Pere : %s bien recu\n", buf);
   }
  return 0;  }
```

Exemple 2

Ce second exemple fait intervenir les signaux. Il fonctionne selon le scénario suivant :

- le fils ferme les deux côtés du tube
- le père essaye d'écrire dedans : il reçoit alors un signal SIGPIPE qu'il va intercepter.
- le fonction de traitement envoie le signal SIGTERM au fils
- le père se bloque sur la fonction wait (et attend de recevoir un SIGCHILD)
- le fils intercepte le SIGTERM et affiche simplement un message avant de s'arrêter grâce à la fonction exit
- le père étant débloqué, il écrit un message puis s'arrête.

Exemple 2

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

void trtSIGPIPE (int signalRecu, siginfo_t* info, void* pasUtileIci)
 {
  printf("Je receptionne le signal %d envoye par le processus %d\n",
         signalRecu, info->si_pid);
  printf("J'envoie le signal SIGTERM à ce processus\n");
  kill (info->si_pid, SIGTERM);
  fflush(stdout);
 }

void trtSIGTERM (int signalRecu, siginfo_t* info, void* pasUtileIci)
 {
  printf("Je receptionne le signal %d envoye par le processus %d\n",
         signalRecu, info->si_pid);
  printf("Je m'arrête avec la valeur 50\n");
  fflush(stdout);
  exit (50);
 }
```

Exemple 2

```
int main(int argc, char** argv)
 {
  int   tube[2];
  pid_t pid;
  int   etat;

  struct sigaction pourSIGPIPE;
                   pourSIGPIPE.sa_sigaction=&trtSIGPIPE;
                   pourSIGPIPE.sa_flags=SA_SIGINFO;

  sigemptyset(&pourSIGPIPE.sa_mask);
  sigaction(SIGPIPE, &pourSIGPIPE, NULL);

  struct sigaction pourSIGTERM;
                   pourSIGTERM.sa_sigaction=&trtSIGTERM;
                   pourSIGTERM.sa_flags=SA_SIGINFO;

  sigemptyset(&pourSIGTERM.sa_mask);
  sigaction(SIGTERM, &pourSIGTERM, NULL);

  pipe(tube);
```

Exemple 2

```
switch (fork())
 {
  case -1 : printf ("Erreur de creation du processus fils.\n");
            perror ("Erreur");
            break;

  case  0 : printf ("Fils : PID=%d\n", getpid());
            close(tube[0]);
            close(tube[1]);
            printf ("Fils : tout est ferme\n");
            while(1);

  default : printf ("Pere : PID=%d\n", getpid());
            close(tube[0]);
            sleep (1) ;
            write(tube[1], "Bonjour", 8);
            pid = wait(&etat);
            printf("Pere : fils %d s'arrete avec la valeur %d\n",
                    pid, &etat);
  }
 return 0;
}
```

La différence

Comme les tubes anonymes, les tubes nommés sont des zones de données organisées en FIFO

Les tubes anonymes sont détruits lorsque le processus qui les a créés disparaît alors que les tubes nommés sont liés au système d'exploitation et ils doivent être explicitement détruits.

La fonction mkfifo

La création d'un tube nommé s'effectue grâce à la fonction mkfifo qui nécessite d'inclure les fichiers d'entête sys/types.h et sys/stat.h

```
int mkfifo (const char *pathname, mode_t mode)
```

Le premier argument est le chemin d'accès vers le fichier
Le second correspond aux permissions d'accès Unix (voir page 90)

Remarque : On peut aussi le faire la fonction mknod mais cela est non recommandé par la norme POSIX car cette fonction est non portable

Voir http://www.lefinnois.net/artPROG/Pipes/tubes.php

La fonction mkfifo

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans le second cas, errno peut être égal à :

- EACCES : un des répertoires du chemin d'accès ne peut pas être exploré (problème de permission)
- EEXIST : le fichier existe déjà
- ENAMETOOLONG : le nom du fichier est trop long
- ENOENT : un des répertoires du chemin d'accès n'existe pas
- ENOSPC: il n'y a plus assez de place sur le système de fichiers

La fonction open

Une fois le tube nommé créé, on l'ouvre comme un fichier grâce à la fonction open (il faut inclure sys/stat.h et fcntl.h) dont la signature est la suivante :

```
int open(const char *fic,int type,mode_t droits)
```

Les arguments utilisés sont les suivants :

- le nom du fichier (le nom du tube dans notre cas)
- le type d'ouverture (décrit juste après)
- les droits d'accès Unix à ce fichier

La fonction open

Le deuxième argument est un entier formé en faisant des OU logique entre les constantes symboliques ci-dessous (open n'étant pas spécifique aux tubes, certaines options ne sont pas utiles dans notre cas) :

- **0_RDONLY** : lecture seule
- **0_WRONLY** : écriture seule
- **0_RDWR** : lecture et écriture
- 0_APPEND : écriture à la fin de fichier
- 0_TRUNC : supprimer les données existantes lors de l'ouverture
- 0_CREAT : créer le fichier si nécessaire
- 0_EXCL : vérifier si le fichier existe déjà lors de la création

La fonction open

Cette fonction renvoie un entier correspond au descripteur de fichier en cas de succès. En cas d'échec, open retourne -1 et errno peut alors valoir :

- EEXIST : le fichier qu'on cherche à créer (option O_CREAT) existe déjà (test effectué lorsque l'option O_EXCL est activée)
- EISDIR : on essaye d'écrire dans un fichier de type répertoire
- EACCES : problème de droit d'accès au fichier ou à un répertoire du chemin d'accès à ce fichier
- ENAMETOOLONG : le nom du fichier est trop long

La fonction open

- ENOENT : un des répertoires du chemin d'accès n'existe pas
- ENOTDIR : un des éléments du chemin d'accès n'est pas un répertoire ou le chemin donné en paramètre ne correspond pas à un répertoire alors que l'option O_DIRECTORY est activée
- ENXIO : Les options O_NONBLOCK | O_WRONLY sont activées pour écrire dans un tube nommé, mais il n'y a pas de processus pour le lire.
- ENODEV : le chemin d'accès correspond à un fichier spécial, mais il n'y a pas le périphérique correspondant.

La fonction open

- EROFS : On veut écrire dans un fichier qui se trouve dans une partie du filesystem en lecture seule.
- ETXTBSY : On veut écrire dans un fichier exécutable en cours d'utilisation.
- EFAULT : le pointeur pathname (premier argument) n'est pas bon
- ELOOP : le chemin d'accès contient une référence circulaire via un lien symbolique ou l'option O_NOFOLLOW est activée alors que le chemin d'accès un lien symbolique.

La fonction open

- ENOSPC : le fichier correspondant au chemin d'accès pathname devrait être créé (option O_CREATE), mais il n'y a plus assez de place sur le périphérique concerné.
- ENOMEM : Il n'y a plus assez de mémoire pour le noyau
- EMFILE : Le nombre de fichiers ouverts par le processus a atteint la limite maximale
- ENFILE : Le nombre total de fichiers ouverts pour l'ensemble du système a atteint la limite maximale.

Les fonctions read et write

L'écriture et la lecture se font grâce aux fonctions write et read présentées précédemment :

```
ssize_t read(int desc, void *tampon, size_t nbOctets)

ssize_t write(int desc, const void *tampon, size_t nbOctets)
```

Le premier paramètre est le descripteur de fichier (ici le tube nommé).

Le deuxième est le pointeur vers le tampon qui va recueillir les données reçues ou qui contient les données à expédier.

Le dernier est le nombre d'octets à récupérer (au plus) ou à expédier.

La fonction close

L'accès au tube est fermé grâce à la fonction close (qui prend en paramètre le descripteur).

Cette fonction est définie dans unistd.h et a comme prototype :

```
int close(int fildes)
```

La fonction unlink

Le tube (le fichier) doit aussi être détruit grâce à la fonction unlink définie dans unistd.h de la manière suivante :

```
int unlink(char * lien )
```

Cette fonction prend en paramètre le chemin d'accès au tube à supprimer et retourne 0 ou -1 selon que la suppression s'est déroulée correctement ou non.

Un programme écrivain

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

main (int n, char ** v)
 {
  int p;
  if (mkfifo ( v[1], S_IFIFO | 0666) == -1)
   {
    perror ("creation impossible ");
    exit (1);
   }

  if ((p = open (v[1], O_WRONLY, 0) )== -1) // Bloqué si pas de lecteur
   {
    perror ("ouverture impossible ");
    exit (2);
   }

  write (p, "ABCDEFGH", 8);
 }
```

Un programme lecteur

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

main (int n, char ** v)
 {
  int p;
  char c;

  if ((p = open (v[1], O_RDONLY, 0)) == -1)
   {
    perror ("ouverture impossible ");
    exit (2);
   }

  read (p, &c, 1);
  printf ("%c\n", c);
 }
```

Les sockets

2SBD et Berknet

En 1978, Eric Schmidt (PDG de Google jusqu'en avril 2011) a développé un petit programme réseau sur 2BSD.

Cela a permis par la suite de mettre en relation des machines (PDP-11 et VAX 11) au sein du Berknet : cela a attiré l'attention de la DARPA.

Voir http://books.google.com/books?id=wshm3f0hyI8C&pg=PA498#v=onepage&q&f=false
http://en.wikipedia.org/wiki/Eric_Schmidt

Le développement des sockets

L'organisme a alors financé les travaux de l'Université de Berkeley pour la mise au point d'une API de communication inter-processus qui implantent la nouvelle suite de protocoles TCP/IP.

La 4.2BSD, sortie officiellement en septembre 1983, était le premier Unix à permettre les communications au sein du nouveau réseau Internet (le « flag day » considéré comme la date de naissance de l'Internet étant le 1 janvier 1983).

Les sockets BSD sont donc devenus un standard de fait.

Voir http://books.google.com/books?id=wshm3f0hyI8C&pg=PA498#v=onepage&q&f=false

La structure sockaddr

L'API socket pouvant fonctionner avec divers protocoles (donc avec des adresses très variées), les fonctions, qui la composent, manipulent une structure générique définie de la manière suivante dans sys/socket :

```
struct sockaddr
 {
  u_char   sa_len;        /* Pas toujours présent */
  u_char   sa_family;
  char     sa_data[14];
 };
```

Le premier champ contient la longueur réelle de l'adresse fournie dans le champ sa_data dans une forme qui diffère selon la famille d'adresses utilisée (champ sa_family).

Voir http://www.manpagez.com/man/4/networking/

La structure sockaddr

Le champ sa_familly contient donc un entier correspondant à la famille d'adresses utilisée (les constantes sont définies dans sys/socket) :

- AF_UNSPEC (0) signifie que cette famille est non spécifiée (cela permet, par exemple à des fonctions comme getaddrinfo de renvoyer les adresses de socket de n'importe quel type.
- **AF_UNIX** (obsolète) ou **AF_LOCAL** (1) indique qu'il s'agit de socket du domaine Unix (local)
- **AF_INET** (2) correspond au protocole IPv4

La structure sockaddr

- AF_AX25 (3) désigne un protocole appelé AX25, une adaptation du protocole X25 pour des réseaux hertziens utilisés par des radio amateurs
- AF_IPX (4) correspond au protocole IPX de Novell
- AF_APPLETALK (5) concerne le réseau Apple Talk
- AF_NETROM (6) est associé à un autre protocole de communication par radio
- AF_BRIDGE (7) pour les passerelles multiprotocoles
- AF_ATMPVC (8) concerne les réseaux ATM
- AF_X25 (9) est utilisé pour le protocole X.25

La structure sockaddr

- **AF_INET6** (10) est utilisé pour manipuler des adresses de type IPv6
- AF_ROSE (11) correspond au protocole Rose utilisé dans les radiocommunications
- AF_DECnet (12) permet de manipuler des adresses utilisées dans les réseaux DECnet
- AF_NETBEUI (13)
- AF_SECURITY (14)
- AF_KEY (15)
- AF_NETLINK ou AF_ROUTE (16)

La structure sockaddr

- AF_PACKET (17)
- AF_ASH (18)
- AF_ECONET (19)
- AF_ATMSVC (20)
- AF_SNA (22)
- AF_IRDA (23)
- AF_PPOX (24)
- AF_WANPIPE (25)
- AF_MAX (32)

La structure associée aux sockets Unix

La famille AF_UNIX permet de créer des sockets entre deux processus Unix se trouvant sur la même machine. Pour cela, on utilise la structure ci-dessous :

```
struct sockaddr_un
 {
  uint8_t      sun_len;     /* BSD 4.4 et Unix 98 */
  sa_family_t  sun_family;  /* AF_UNIX ou AF_LOCAL */
  char         sun_path[126];
 };
```

Les deux premiers champs proviennent de la structure générique sockaddr. La dernière est le chemin vers une entrée dans le système de fichiers.

La communication passe par un fichier

Cette entrée commune est comparable à un tube par lequel transitent les informations. La différence est qu'elle est bidirectionnelle.

```
struct sockaddr_un adresse;

adresse.sun_family = AF_UNIX;
strcpy(adresse.sun_path, "/tmp/fichierCommun");
adresse.sun_len = strlen (adresse.sun_path) +
                  sizeof (adresse.sun_len) +
                  sizeof (adresse.sun_family);
```

Voir http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzab6%2Fcafunix.htm
http://www.minek.com/files/unix_examples/poll.html
http://beej.us/guide/bgipc/output/html/multipage/unixsock.html
http://www.cis.temple.edu/~ingargio/old/cis307s96/readings/docs/ipc.html

La création de la socket

Cette fonction permet de créer la structure de données qui va gérer le socket. Elle est définie dans sys/socket.h et a la signature suivante :

```
int socket(int domain, int type, int protocol);
```

Le premier argument correspond au domaine sur lequel va être utilisé le socket. Ce domaine est désigné grâce à une constante symbolique de type « PF_ » qui est en réalité un alias des constantes symboliques de type « AF_ ».

Dans le cas présent, on écrira donc « PF_UNIX » (obsolète) ou « PF_LOCAL ».

La création de la socket

Le deuxième argument permet de spécifier le type de socket à mettre en place grâce aux constantes symboliques suivantes :

- SOCK_STREAM : la socket Unix n'impose pas de limites, car les données sont envoyées dans un flux d'octets.
- SOCK_DGRAM : la socket Unix limite la taille des données envoyées à des blocs de 1500 octets (en général)
- SOCK_RAW, SOCK_SEQPACKET et SOCK_RDM qui ne sont pas utiles dans ce cas de figure.

Voir http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzab6%2Fcafunix.htm

La création de la socket

Le dernier argument est utilisé pour spécifier le protocole de transport, mais cet élément n'a pas de sens dans le cas des sockets Unix. On le met donc généralement à 0.

La fonction socket renvoie un entier qui peut être :

- Un entier strictement positif correspond au descripteur
- -1 pour indiquer que le socket n'a pas pu être créé

Voir http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzab6%2Fcafunix.htm

Les erreurs possibles

Dans le second cas de figure, errno est égal à l'une des valeurs ci-dessous :

- EACCES : on n'a pas les droits nécessaires pour créer le socket avec le type et/ou le protocole spécifié
- EAFNOSUPPORT : l'implémentation de l'API socket ne gère pas la famille d'adresse spécifiée
- EINVAL : protocole inconnu ou famille de protocoles non disponible
- EMFILE : la table des fichiers du processus est pleine (trop de descripteurs de fichiers utilisés)

Les erreurs possibles

- ENFILE : le nombre total de fichiers qui peuvent être ouverts par le système d'exploitation a été atteint
- ENOBUFS or ENOMEM : il n'y a plus assez de mémoire pour créer le socket
- EPROTONOSUPPORT : le type de protocole ou le protocole spécifié n'est pas supporté au sein du domaine qui a été mentionné en paramètre.

Voir http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp?topic=%2Frzab6%2Fcafunix.htm
http://books.google.fr/books?id=Y7KOyI8knWQC&pg=PA173

Mode connecté versus mode datagramme

Pour créer un socket Unix sans limitation sur la taille des données, on écrit :

```
int descripteur = socket (PF_LOCAL, SOCK_STREAM, 0);
```

Pour créer un socket Unix qui limite la taille des données envoyées, on écrit :

```
int descripteur = socket (PF_LOCAL, SOCK_DGRAM, 0);
```

La création du point de connexion

Cette fonction permet de lier le socket à un point de connexion géré par le système d'exploitation. Elle est définie dans sys/socket.h et a la signature suivante :

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

Le premier argument est le descripteur retourné par la fonction socket.

Les deux autres arguments sont l'adresse de la structure de type sockaddr_un et sa taille en octets.

Voir http://www.kernel.org/doc/man-pages/online/pages/man2/bind.2.html

La création du point de connexion

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans ce second cas, errno est égal à l'une des valeurs ci-dessous :

- EACCES : l'adresse est protégée (le numéro de port est inférieur à 1024) et l'utilisateur n'a pas les droits de root
- EADDRINUSE : l'adresse (le numéro de port) est déjà utilisée
- EBADF : le descripteur de socket n'est pas valide
- EINVAL : le socket est déjà lié à un point de connexion
- ENOTSOCK : le descripteur passé en paramètre est un descripteur de fichier et non un descripteur de socket

Les erreurs possibles

Les erreurs décrites ci-dessous sont spécifiques aux sockets UNIX (domaine AF_UNIX ou AF_LOCAL) :

- EACCES : L'exploration de certains répertoires contenus dans le chemin d'accès au fichier est interdite
- EADDRNOTAVAIL : On essaye d'utiliser une interface (fichier) qui n'existe pas ou une adresse qui n'est pas locale.
- EFAULT : l'adresse se situe en dehors de l'espace d'adressage accessible par l'utilisateur.
- EINVAL : la longueur de l'adresse n'est pas bonne ou le socket n'est pas de la famille AF_UNIX.

Les erreurs possibles

- ELOOP : le chemin d'accès nécessite de passer par trop de liens symboliques lors de sa résolution.
- ENAMETOOLONG : l'adresse est trop longue
- ENOENT : le fichier n'existe pas
- ENOMEM : il n'y a pas assez de mémoire.
- ENOTDIR : une partie du chemin d'accès ne fait pas référence à un répertoire
- EROFS : le fichier utilisé par le socket est situé dans une zone en lecture seule (l'écriture n'est pas possible dans ce fichier).

La mise en mode écoute

Cette fonction prépare le socket au mode « écoute » et crée à cet effet une file d'attente des requêtes. Elle est définie dans sys/socket.h et a la signature suivante :

```
int listen (int socket, int backlog);
```

Le premier argument de cette fonction est le descripteur retourné par la fonction socket et le second argument correspond à la taille de la file d'attente (le nombre de requêtes qui peuvent être mises en attente).

Voir http://www.manpagez.com/man/2/listen/

Les erreurs possibles

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans ce second cas, errno est égal à l'une des valeurs ci-dessous :

- EACCES : Le processus ne dispose pas des droits pour exécuter la fonction listen sur ce socket
- EBADF : Le premier argument n'est pas un descripteur de socket valide.
- EDESTADDRREQ : Le socket n'est pas lié à un point de connexion et le protocole ne supporte pas l'écoute sur un socket non lié.
- EINVAL : Le socket est déjà connecté

Les erreurs possibles

- ENOTSOCK : Le premier argument (le descripteur de fichier) ne désigne pas un socket
- EOPNOTSUPP : Le type de socket est tel qu'il ne supporte pas la fonction listen

Le serveur se bloque en attente d'une connexion

Cette fonction, définie dans sys/socket.h, bloque le serveur en attente d'une connexion. Sa signature est la suivante :

```
int accept(int socket, struct sockaddr *restrict address,
           socklen_t *restrict address_len);
```

Le premier argument est le descripteur de la socket serveur. Les deux autres arguments sont l'adresse du socket (cf. page 194) et la longueur de cette adresse en octet.

Cette fonction retourne -1 en cas d'erreur ou le descripteur de la socket utilisée pour dialoguer avec le client.

Voir http://www.manpagez.com/man/2/accept/

Les erreurs possibles

Si la fonction renvoie -1, errno peut alors être égal à l'une des constantes ci-dessous :

- EBADF : le premier paramètre n'est pas un descripteur de socket valide
- ECONNABORTED : la connexion a été abandonnée par le client
- EFAULT : le deuxième argument n'est pas valide (l'adresse n'est pas correcte)
- EINTR : la fonction accept a été interrompue par un signal
- EINVAL : la socket n'est pas en attente de connexion ou le troisième argument est invalide.

Les erreurs possibles

- EMFILE : la table des descripteurs du processus est pleine
- ENFILE : la table des descripteurs du système est pleine
- ENOMEM : mémoire insuffisante pour effectuer le traitement
- ENOTSOCK : le premier argument désigne autre chose qu'une socket
- EOPNOTSUPP : la socket ne supporte pas la fonction accept car elle n'est pas de type SOCK_STREAM
- EWOULDBLOCK : la socket est non-bloquante et il n'y a aucune demande de connexion

La lecture et l'écriture dans la socket

L'écriture et la lecture peuvent se faire grâce aux fonctions write et read présentées précédemment :

```
ssize_t read(int desc, void *tampon, size_t nbOctets)

ssize_t write(int desc, const void *tampon, size_t nbOctets)
```

Le premier paramètre est le descripteur de fichier (ici la socket).

Le deuxième est le pointeur vers le tampon qui va recueillir les données reçues ou qui contient les données à expédier.

Le dernier est le nombre d'octets à récupérer (au plus) ou à expédier.

Une fonction spécifique d'envoi

L'envoi d'information peut aussi s'effectuer grâce à la fonction send qui est définie dans sys/socket.h

```
ssize_t send (int socket, const void *buffer,
              size_t length, int flags);
```

Le premier paramètre est le descripteur de socket.

Le deuxième argument est l'adresse du tampon contenant les données à envoyer. Le troisième argument est le nombre d'octets à envoyer.

Une fonction spécifique d'envoi

Le dernier argument est un entier qui peut être un OU logique entre les constantes énumérées ci-dessous mais il est souvent mis à 0.

Pour les sockets Unix, on peut utiliser les valeurs suivantes :

- MSG_DONTWAIT : pour un fonctionnement non bloquant de send (peut aussi être obtenu grâce à la fonction fcntl)
- MSG_NOSIGNAL : ne pas générer de signal SIGPIPE si le client a fermé sa socket (coupé la connexion)

Les erreurs possibles

send renvoie le nombre de caractères émis, ou -1 si elle échoue, errno prend alors une des valeurs ci-dessous :

- EACCES : dans le cas d'une socket Unix, on ne dispose pas des droits d'écriture sur le fichier tampon
- EAGAIN : la socket est non bloquante mais l'envoi n'est pas possible (cet envoi aurait dû être bloqué)
- EBADF : le descripteur de socket est invalide
- ECONNRESET : connexion réinitialisée par le client
- EDESTADDRREQ : la socket est de type connectée mais aucune adresse de destination n'a été spécifiée
- EFAULT : le pointeur désignant le tampon n'est pas valide

Les erreurs possibles

- EINTR : la fonction send a été interrompue par la réception d'un signal
- EINVAL : l'un des arguments (par exemple, le descripteur de socket) a une valeur incorrecte
- EISCONN : on veut connecter une socket qui est déjà connectée à une autre adresse
- EMSGSIZE : la taille du message envoyée sur la socket est supérieure à ce qu'elle peut supporter
- ENOBUFS : le tampon d'émission de la socket est pleine
- ENOMEM : il n'y a plus assez de mémoire
- ENOTCONN : la socket n'est pas connectée
- ENOTSOCK : le premier paramètre ne désigne pas une socket

Les erreurs possibles

- EOPNOTSUPP : l'une des options de l'argument flag n'est pas supporté par ce type de socket
- EPIPE : le correspondant est absent (un signal SIGPIPE est générée sauf si l'option MSG_NOSIGNAL est activée)

Une fonction spécifique de réception

Cette fonction est le pendant de la fonction send pour la réception d'information. Elle est définie dans sys/socket.h

```
ssize_t recv (int socket, const void *buffer,
              size_t length, int flags);
```

Le premier paramètre est le descripteur de socket.

Le deuxième argument est l'adresse du tampon qui va accueillir les données reçues. Le troisième argument est le nombre d'octets à récupérer.

Une fonction spécifique de réception

Le dernier argument permet de spécifier les drapeaux de fonctionnement à utiliser au moyen d'un OU logique entre des constantes symboliques (cet argument est souvent à 0).

On retrouve la constante MSG_DONTWAIT utilisée pour la fonction send avec les sockets Unix. On peut aussi utiliser :

- MSG_PEEK : pour les données sans les retirer de la socket
- MSG_WAITALL : pour rendre l'opération de lecture bloquante jusqu'à ce que toute les données soient récupérées (sauf s'il y a une déconnexion ou l'arrivée d'un signal)

Les erreurs possibles

recv renvoie le nombre de caractères reçus, ou -1 si elle échoue, errno prend alors une des valeurs ci-dessous :

- EAGAIN : la socket est non bloquante mais il n'y a pas de données à lire (cette réception aurait dû être bloquée)
- EBADF : le descripteur de socket est invalide
- EFAULT : le pointeur désignant le tampon n'est pas valide
- EINTR : la fonction send a été interrompue par la réception d'un signal
- EINVAL : l'un des arguments (par exemple, le descripteur de socket) a une valeur incorrecte
- ENOMEM : il n'y a plus assez de mémoire

Les erreurs possibles

- ENOTCONN : la socket n'est pas connectée
- ENOTSOCK : le premier paramètre ne désigne pas une socket

Fermeture de la socket

Lorsque la socket n'est plus utile, elle doit être fermée à l'aide de la fonction close, définie dans unistd.h afin de libérer les ressources mémoires. Sa signature est la suivante :

```
int close(int fd);
```

Cette fonction prend le descripteur de socket en paramètre et renvoie un entier qui est mis à 0 en cas de succès et à -1 en cas d'échec.

Les erreurs possibles

Si la fermeture a échoué, errno peut être égal à l'une des constantes ci-dessous :

- EBADF : le descripteur de socket n'est pas valide
- EINTR : la fonction close a été interrompue par un signal
- EIO : une erreur d'entrée-sortie s'est produite.

Exemple

Voici un exemple de code pour le serveur.

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

int main (int argc, char** argv)
 {
  int sockfd, newsockfd;
  socklen_t clilen, servlen;

  struct sockaddr_un cli_addr;
  struct sockaddr_un serv_addr;

  char tampon [30];
  int nbOctets;

  if ( (sockfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
   {
    printf ("Erreur de creation de socket\n");  exit (1);
   }
```

Exemple

```
bzero((char *) &serv_addr, sizeof(serv_addr));
serv_addr.sun_family = AF_UNIX;
strcpy(serv_addr.sun_path, "/tmp/socketLocale.1");
servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);

if ( bind (sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
 {
  printf ("Erreur de bind\n");    exit (1);
 }

listen(sockfd, 5);

while (1)
 {
  clilen = sizeof(cli_addr);

  printf ("serveur: En attente...\n");
  newsockfd = accept (sockfd, (struct sockaddr *) &cli_addr, &clilen);

  if (newsockfd < 0)
   printf ("serveur: Erreur de accept\n");

  nbOctets = 0;
  bzero ((char *) tampon, 30);
```

Exemple

```
    read (newsockfd, tampon,    20);
    printf ("Serveur reçoit : %s\n", tampon);

    write (newsockfd, "Message reçu", 13);
    printf ("Serveur envoie : Message recu\n");

    close(newsockfd);
   }

 return 0;
}
```

Plus simple à implémenter

Le code du **client** est plus simple :

- on crée une structure de type `sockaddr_un` (qui contient les mêmes informations) ;
- on crée une socket avec l'instruction `socket` afin de pouvoir dialoguer avec le serveur ;
- à la différence du serveur, on utilise l'instruction **`connect`** pour initier un dialogue avec le serveur (le serveur, en attente sur l'instruction `accept` est débloqué) ;
- on utilise les instructions `read`, `write` ou `send`, `recv` pour envoyer et recevoir des informations ;
- on termine la communication en fermant la socket avec l'instruction `close`.

Exemple

Cette fonction permet à un client de se connecter au serveur afin d'échanger des informations par l'intermédiaire des sockets. Elle est définie dans sys/socket.h et sa signature est :

```
int connect (int sockfd, const struct sockaddr *addr,
                         socklen_t addrlen);
```

Le premier paramètre est le descripteur de socket renvoyée par la fonction socket.

Les deux autres paramètres sont le pointeur sur la structure de type sockaddr_un (à caster), qui contient les informations de connexion, et la taille de cette structure en octets.

Exemple

Cette fonction renvoie 0 en cas de succès et -1 en cas d'échec. Dans ce second cas, la variable errno peut être égal à l'une des constantes ci-dessous :

- EACCES : Le processus ne dispose pas des droits pour exécuter la fonction listen sur ce socket
- EADDRINUSE: L'adresse locale est déjà utilisée.
- EAFNOSUPPORT : L'adresse transmise n'a pas la bonne valeur dans son champ sa_family.
- EAGAIN : Pas de port disponible
- EALREADY : La socket est non bloquante et une connexion précédente ne s'est pas encore terminée
- EBADF : le descripteur de socket est invalide

Exemple

- ECONNREFUSED : La connexion est refusée par le serveur.
- EFAULT : Le pointeur désignant le tampon n'est pas valide
- EINTR : La fonction send a été interrompue par la réception d'un signal
- EISCONN : La socket est déjà connectée
- ENOTSOCK : le premier paramètre ne désigne pas une socket
- ETIMEDOUT : Le délai pour se connecter est dépassé

Exemple

Voici un exemple de code pour le client.

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

int main (int argc, char** argv)
 {
  int             sockfd, servlen;
  struct sockaddr_un serv_addr;

  char tampon [30];
  int nbOctets;

  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sun_family = AF_UNIX;
  strcpy(serv_addr.sun_path, "/tmp/socketLocale.1");
  servlen = strlen(serv_addr.sun_path) + sizeof(serv_addr.sun_family);
```

Exemple

```
 if ((sockfd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
  {
   printf ("Erreur de creation de socket\n");    exit (1);
  }

 if ( connect(sockfd, (struct sockaddr *) &serv_addr, servlen) < 0)
  {
   printf ("Erreur de creation de connect\n");
   exit (1);
  }

 write (sockfd, "Bonjour le serveur !", 20);
 printf ("Client envoie : Bonjour le serveur !\n");

 nbOctets = 0;
 bzero ((char *) tampon, 30);

 read (sockfd, tampon,    13);
 printf ("Client reçoit : %s\n", tampon);

 close(sockfd);
 return 0;
}
```

Schéma de synthèse

Pour terminer cette description des sockets, voici un schéma de synthèse.