Labo-Unix
Communication Inter-Processus

Labo-Unix - http ://www.labo-unix.net

2001-2002

Table des matières

1 Manipulation de processus **3**

2 Les pipes **4**

3 Les signaux **5**

4 Les files de messages **6**

5 Sémaphores selon System V **7**

6 Mémoire partagée selon System V **11**

7 Projection en mémoire façon BSD **14**

Introduction

Un processus est un programme en cours d'exécution. Sous Unix, chaque processus possède un identifiant unique, le processus ID(*PID*) dit "pid" à la française, "pi aie di" pour les gens 'aware'. Vous pouvez obtenir la liste des processus tournants et leur pid correspondant à un instant t grâce à la commande 'ps ax'. On aperçoit que la génération du pid d'un processus est simple : elle est incrémentielle. Le premier programme lancé une fois le noyau chargé (le programme 'init' normalement) aura le chiffre 1 pour pid, le second 2 etc ... Les processus appartiennent généralement a l'utilisateur qui les a lancés et héritent donc de ses droits sauf si le fichier exécutable possède l'attribut SUID(SaveUID) ; dans ce cas, le processus récupère les droits du propriétaire du fichier. Nous allons voir dans ce cours comment, à partir d'un seul programme, on peut créer une application dite 'multitache'.

1 Manipulation de processus

Il existe plusieurs fonctions qui permettent de lancer des commandes à partir d'un programme C :

```
- int system(char *string)
```

La chaine prise en argument est le nom d'un programme, d'un script shell exécutable ou d'une commande que l'on souhaite lancer.

La valeur de retour est le statu de sortie du shell ayant lancé cette commande. D'une manière générale, un chiffre différent de 0 signifie qu'une erreur s'est produite.

system() est composé de trois appels systèmes que nous allons décrire : *execl*(), *wait*() et *fork*()

```
- execl(char *path, char *arg0, ..., char *argn, 0)
```

execl() veut dire "EXECute and Leave". Cette fonction va donc créer un nouveau processus qui va remplacer le processus courant en mémoire.

Le dernier paramètre doit toujours être 0.

path est le chemin complet du fichier exécutable qu'on souhaite lancer.

arg0 est le nom de la commande (donc le nom du fichier) et *arg1* à *argn* sont les arguments pour la commande.

```
- int fork()
```

C'est une fonction très utile qui permet, à partir d'un processus (père), de créer un deuxième processus identique (fils).

En cas de succès, *fork*() va renvoyer 0 au processus fils et le pid du processus fils au père.

En cas d'erreur, aucun fils n'est créé et *fork*() renvoie -1.

Cette fonction va donc permettre de créer des programmes multitaches simples.

```
Exemple :
#include <stdio.h>
#include <unistd.h>
int main(void){
  int pid;
  printf("Création d'un processus fils\n");
  pid = fork(); // Création de la copie du  processus cou-
rant (fils)
    if (pid < 0) { // Si fork() a échoué
      printf("Erreur : fork()!\n");
      return -1;
    }
    if (pid == 0) { // Si on est dans le processus fils ...
      printf("Je suis le fils\n");
    }
```

```
    else{ // Si on est dans le père
      printf("Je suis le père\n");
    }
  return 0;
}
```

Lorsqu'on exécute plusieurs fois ce programme, on s'aperçmêmeoit que l'ordre d'affichage des deux phrases peut changer. Ceci est du au fait que les deux processus fonctionnent en même temps sur le système. C'est le système (le noyau) qui ordonnance les différents processus et leur attribue du temps CPU. Cet ordonnancement dépend de nombreux paramètres (notamment de l'utilisation par d'autres processus du CPU) qui font qu'il est difficile de prévoir quel processus obtiendra du temps CPU avant l'autre. Un autre problème est l'absence de données communes aux deux processus car l'ensemble des variables est recopié en mémoire ; lorsqu'un processus modifie une variable, cette même variable reste intacte dans l'autre processus (seul le nom des variables reste le même, physiquement, elles existent à deux emplacements mémoire différents). Nous verrons plus tard quels sont les moyens qui permettent de contourner ces problèmes.

```
- int wait(int *status_location)
```

Cette fonction force un processus parent à attendre la fin d'un processus enfant ou l'envoi d'un signal à intercepter.

La valeur de retour est le pid du fils et la variable *status_location* est remplie avec le status de sortie de celui-ci.

```
- unsigned int sleep(unsigned int nb_sec)
```

Permet d'endormir un processus pour une durée de *nb_sec* secondes.

La valeur renvoyée est 0 si le temps s'est écoulé ou le nombre de secondes restantes si un signal a réveillé le processus.

```
- exit(int status)
```

Termine le processus appelant cette fonction qui retourne aussi son status de sortie. Cette valeur peut êrécupéréetre récupérée par d'autres processus ou bien par le shell dans la variable "$?".

Les fichiers entêtes permettant d'accéder à ces fonctions sont *unistd.h et stdlib.h* (*sys/types.h* et *sys/wait.h* pour la fonction *wait*()) .

Nous avons donc vu comment on crée plusieurs processus à partir d'un programme. Maintenant, nous allons étudier différentes façons de les faire communiquer entre eux. Ces moyens de communication permettront d'échanger des informations entre les processus mais surtout, il permettront de les contrôleur plus finement. Nous avons à disposition plusieurs méthodes pour accomplir ceci :

- les pipes
- les signaux
- les files de messages
- les sémaphores
- la mémoire partagée
- les sockets (non présentés dans ce cours)

2 Les pipes

Le pipe est un mécanisme qui prend la sortie d'un processus comme entrée d'un autre et inversement. Il existe deux façon d'ouvrir un pipe. L'une est une ouverture formatée, l'autre une ouverture bas niveau.

- ouverture formatée :

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

ouvre un pipe dans lequel 'command' sera un programme connecté au processus appelant.

Le type est soit *r* pour la lecture, soit *w* pour l'écriture.

La valeur de retour est un pointeur sur un flux ou *NULL* en cas d'echec.

On utilise ensuite *fprintf*() et *fscanf*() pour communiquer avec le programme à travers ce tuyau.

exercice 1 : Ecrire un programme utilisant *popen*() et le programme mail pour envoyer un email à votre compte sur votre poste.

- ouverture bas niveau :

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

Cette fonction crée un pipe et renvoie deux descripteurs de fichiers. *fd*[0] est ouvert en mode lecture et *fd*[1] est ouvert en mode écriture.

La valeur de retour est 0 en cas de réussite, -1 en cas d'echec et la variable errno est mise à jour.

Un pipe de ce type va permettre à deux processus crées avec un *fork*() de communiquer en utilisant *read*() et *write*().

Les pipes sont bien pratiques mais une fois de plus, les deux processus ne sont pas forcement parfaitement synchronisés ce qui implique l'utilisation de *wait*() afin que le père attende son fils ; sans cela, on risque l'embrouille familial ... En effet, si le père est en avance sur le fils, il va lire son propre message dans le pipe car il aura appelé *read*() avant son fils.

exercice 2 : Ecrire un programme qui crée deux processus avec *fork*() et dont chaque processus affiche une phrase envoyée par l'autre.

3 Les signaux

Les signaux sont une autre forme de communication entre processus. Il sont utilisés pour rendre compte à un processus d'un événement ou d'une erreur. Il peuvent être générés à la suite d'un événement software (CTRL-C,violation de segment) ou hardware (erreur de bus,périphérique non prêt). Il existe différents signaux pré-définis par le système lui-même qui provoquent la terminaison du processus si celui-ci ne prends fait rien à la réception d'un tel signal. Les comportements par défaut des signaux sont les suivants :

- Le signal est ignoré après avoir été reçu.
- Le processus est terminé après la réception.
- Un fichier core sera écrit puis le processus se terminera.
- Le processus se paralyse à la réception du signal.

La création d'un fichier *core* donnera toutes les informations nécessaires pour permettre d'étudier avec *gdb* le moment précis ou le processus à recu ce signal. Il y a 32 signaux définis, certains peuvent êtres interceptés et pris en charge par le processus d'autres ne peuvent être intercepté ni ignorés. L'ensemble des signaux définis par le système linux se trouvent dans */usr/include/bits/signum.h*.

La fonction qui permet d'envoyer un signal à un processus est *kill*() est l'équivalent de la commande kill sous Unix :

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

Si *pid* est positif, *sig* est envoyé au processus *pid*.

Si *pid* est nul, *sig* est envoyé à tous les processus appartenant au même groupe que le processus appelant.

Si *pid* vaut -1 le signal est envoyé à tous les processus sauf le premier (*init*).

Si *pid* est inférieur à -1 le signal est envoyé à tous les processus du groupe -*pid*.

La fonction renvoie 0 en cas de réussite, -1 en cas d'echec et la variable 'errno' est mise à jour avec le code d'erreur.

Pour pouvoir gérer un certain signal, il faut lui créer un *signal handler* qui va l'intercepter et appeler une fonction préalablement définie.

```
int (*signal(int sig, void (*func)()))()
```

Prend en paramètre le signal *sig* et appelle la fonction pointée par *func*.

La valeur de retour est un pointeur sur *func* en cas de réussite ou -1 si une erreur est survenue.

func peut être un pointeur vers une fonction par défaut (*SIG_DFL*()) qui va terminer le processus ou un pointeur sur une fonction qui ignore le signal (*SIG_IGN*()) sauf si le signal est *SIGKILL* ou enfin un pointer sur une fonction définie par l'utilisateur.

Le fichier entêtes permettant d'accéder à ces fonctions est *signal.h*

exercice 3 : Créez un programme qui crée un processus fils entrant dans une boucle infinie et qui tue ce processus lorsqu'on tape *"kill"*.

exercice 4 : Créez un programme qui, lorsque l'utilisateur effectue un *CTRL-C*, affiche une phrase de mise en garde lui demandant de presser *'Entrée'* pour continuer ou à nouveau *CTRL-C* pour quitter.

4 Les files de messages

C'est un autre mécanisme qui permet l'échange de messages entre processus. Le fonctionnement est différent de celui des pipes car nous n'avons plus affaire à un flux mais à des messages de taille fixe. Les files sont en outre plus souples que les pipes, notamment pour la transmission sélective des messages (c.a.d. pour choisir quel processus doit recevoir un message).

Un message est composé d'un tableau de caractères et d'un entier long.

L'entier long est le type du message. Il servira à déterminer de quel type d'information le corps du message se compose ; il servira aussi à déterminer si un processus doit ou non le récupérer.

Le tableau de caractère représente le corps du message.

Les files de messages peuvent être de deux types : bloquantes ou non bloquantes.

Une file de messages est définie par une structure *msqid_ds* qui est allouée et initialisée lors de sa création.

Certains champs peuvent êtres modifiés en appelant *msgctl*.(Voir *info ipc* pour la définition de la structure).

La fonction permettant de créer une file de messages est *msgget*() :

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

la valeur de retour est un identifiant (*msqid*) créé à partir de *key* ; c'est cet identifiant qui va permettre à différents processus de partager une même file.
msgflg est un entier représentant les différentes options (combinées grâce à un ou logique |).
man msgget pour le detail des options.

La fonction permettant de modifier les attribut d'une file est *msgctl*() :

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

Cette fonction permet, entre autres, de modifier les permissions d'une file.

msqid est l'identifiant d'une file existante.

cmd est l'un de ces flags :

IPC_STAT - Remplis la structure pointée par *buf* avec les informations de status de la file. Le processus doit avoir le droit de lecture sur la file.

IPC_SET - Règle le *GID* et l' *UID* du propriétaire, les permissions et la taille de la file. Le processus doit avoir l'*EUID* du propriétaire ou du root.

IPC_RMID - supprime la file. Encore une fois, un *man msgctl* vous donnera les détails de la fonction, notamment ses valeurs de retour.

Pour envoyer et recevoir des messages, il faut créer créer la structure contenant l'entier long et un tableau de caractères de taille fixe comme ceci :

```
struct msg-
buf { long int mtype; char mtext[TAILLE_MAX_DES_MESSAGES]; };
```

Ensuite on utilise les fonctions msgsnd() et msgrcv().

```
- int msg-
snd(int msqid, const void *msgp, size_t msgsz, int msgflg);
- int msgrcv(int msqid, void *msgp, size_t msgsz, long msg-
typ, int msgflg);
```

Ces fonctions servent respectivement à envoyer et recevoir des messages dans la file.

msqid est l'identifiant d'une fie existante.

msgp est un pointeur vers une structure contenant le type de message et son contenu.

msgz est la longueur en octets du message.

msgtyp est le type de message reçu, spécifié dans le champ *mtype* du processus émetteur.

msgflg spécifie l'action à entreprendre dans l'un de ces deux cas :

- Le nombre d'octets dans la file est déjà égal à *msg_qbytes*.

- Le nombre total de messages sur toutes les files du système est égal aux limites de celui-ci.

L'action à prendre est spécifié avec le flag *NOWAIT* qu'on applique à *msgflg* avec un *ou* binaire :
- Si le flag est appliqué, le message ne sera pas envoyé et la fonction reviendra immédiatement avec le code d'erreur *EAGAIN*.
- Si le flag n'est pas appliqué, le processus ayant appelé cette fonction sera bloqué jusqu'à ce que l'un de ces cas arrive :
- La cause du blocage n'existe plus, et dans ce cas le message est envoyé.
- La file à été supprimée, dans ce cas la fonction revoie -1.
- Le processus ayant appelé la fonction reçoit un signal à gérer, dans ce cas le message n'est pas envoyé et la fonction se termine en revoyant le code d'erreur *EINTR*.

Exercice 5 : Créez deux programmes. Le premier doit déterminer la valeur du plus grand entier inférieur à 10000 et envoyer son résultat au second par l'intermédiaire d'une file. Le second doit lancer le premier et afficher le temps qui passe, seconde par seconde en attendant que le résultat arrive.

5 Sémaphores selon System V

D'une manière générale, les sémaphores permettent d'obtenir un accès en exclusion mutuelle à une ressource. Ça c'est la description typique et pas forcement très claire. En fait ça veut simplement dire qu'on va pouvoir, grâce aux sémaphores, empêcher plusieurs processus d'accéder en même temps à une ressource (une zone de mémoire partagée par exemple). Les sémaphores system V sont très gourmands en ressources et sont bien moins efficaces que les sémaphores de la norme POSIX. Nous n'allons donc pas nous attarder trop longtemps mais il est néanmoins intéressant d'y jeter un oeil car il y a encore peu longtemps, *Linux* n'implementait pas la norme POSIX (pour la programmation multitâche). De nombreux programmes utilisent donc encore ces mécanismes. Voici donc une brève description des fonctions utiles :

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

La valeur de retour de cette fonction est un identifiant de jeu de sémaphores (*semid).*

key est un identifiant accès. C'est le même principe que la clef des files de messages.

nsems spécifie le nombre d'éléments dont sera composé le tableau de sémaphores.

semflg représente les option de création et l'accès au jeu de sémaphores.

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

Permet de controler les caractéristiques d'un jeu de sémaphores.

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

semid est l'identifiant de je de sémaphores retourné par *semget*() .

sops est un pointeur sur un tableau de structures. Chaque structure contient un numéro de sémaphore, l'action à effectuer, des flags de contrôle.

l'action à effectuer est soit :

- incrémenter le sémaphore d'une valeur si on spécifie une valeur positive.
- décrémenter le sémaphore d'une valeur si on spécifie une valeur négative. Si on essaye de décrémenter le sémaphore en dessous de zéro, la fonction échoue ou se bloque selon qu'on a spécifié ou non le flag *IPC_NOWAIT* dans les options de contrôle .
- attendre que le sémaphore atteigne la valeur 0 si on a spécifié 0.

Voici un bon exemple de d'utilisation de sémaphores pêché dans une documentation sur System V :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
main()
{
   int i,j;
   int pid;
   int semid; /* semid of semaphore set */
   key_t key = 1234; /* key to pass to semget() */
   int semflg = IPC_CREAT | 0666; /* semflg to pass to semget() */
   int nsems = 1; /* nsems to pass to semget() */
   int nsops; /* number of operations to do */
   struct sembuf *sops = (struct sembuf *) malloc(2*sizeof(struct sembuf)); /* ptr to
   /* set up semaphore */

   (void) fprintf(stderr, "\nsemget: Setting up seamaphore: semget(%#lx, %\
%#o)\n",key, nsems, semflg);
   if ((semid = semget(key, nsems, semflg)) == -1) {
      perror("semget: semget failed");
      exit(1);
   } else
     (void) fprintf(stderr, "semget: semget succeeded: semid =\
%d\n", semid);   /* get child process */

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

   if (pid == 0)
     { /* child */
i = 0;
```

```
while (i < 3) {/* allow for 3 semaphore sets */

   nsops = 2;

   /* wait for semaphore to reach zero */

   sops[0].sem_num = 0; /* We only use one
 track */
   sops[0].sem_op = 0; /* wait for semaphore flag to become zero */
   sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


   sops[1].sem_num = 0;
   sops[1].sem_op = 1; /* increment semaphore -- take control of track */
   sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

   /* Recap the call to be made. */

   (void) fprintf(stderr,"\nsemop:Child  Calling
   semop(%d, &sops, %d) with:", semid, nsops);
   for (j = 0; j < nsops; j++)
     {
(void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j,
       sops[j].sem_num);
(void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
(void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
     }

           /* Make the semop() call and report the results. */
   if ((j = semop(semid, sops, nsops)) == -1) {
      perror("semop: semop failed");
   }
   else
     {
(void) fprintf(stderr, "\tsemop: semop returned
 %d\n", j);

(void) fprintf(stderr, "\n\nChild Process Taking Control of Track: %d/3
 times\n", i+1);
sleep(5); /* DO Nothing for 5 seconds */
nsops = 1;

/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP COntrol of
      track */
sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
  asynchronous  */


if ((j = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
}
```

```
else
  (void) fprintf(stderr, "Child Process
   Giving up Control of Track: %d/3 times\n", i+1);
sleep(5); /* halt process to allow parent to catch semaphor
   change first */
            }
   ++i;
}

     }
   else /* parent */
     {  /* pid hold id of child */

i = 0;



while (i  < 3) { /* allow for 3 semaphore sets */

   nsops = 2;

   /* wait for semaphore to reach zero */
   sops[0].sem_num = 0;
   sops[0].sem_op = 0; /* wait for
semaphore flag to become zero */
   sops[0].sem_flg = SEM_UNDO; /* take off semaphore asynchronous  */


   sops[1].sem_num = 0;
   sops[1].sem_op = 1; /* increment semaphore --
take control of track */
   sops[1].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore */

   /* Recap the call to be made. */

   (void) fprintf(stderr,"\nsemop:Parent Calling
   semop(%d, &sops, %d) with:", semid, nsops);
   for (j = 0; j < nsops; j++)
     {
(void) fprintf(stderr, "\n\tsops[%d].sem_num = %d, ", j,
       sops[j].sem_num);
(void) fprintf(stderr, "sem_op = %d, ", sops[j].sem_op);
(void) fprintf(stderr, "sem_flg = %#o\n", sops[j].sem_flg);
     }

           /* Make the semop() call and report the results. */
   if ((j = semop(semid, sops, nsops)) == -1) {
      perror("semop: semop failed");
   }
   else
     {
(void) fprintf(stderr, "semop: semop returned %d\n", j);

(void) fprintf(stderr, "Parent Process Taking
```

```
 Control of Track: %d/3 times\n", i+1);
sleep(5); /* Do nothing for 5 seconds */
nsops = 1;

/* wait for semaphore to reach zero */
sops[0].sem_num = 0;
sops[0].sem_op = -1; /* Give UP COntrol of
      track */
sops[0].sem_flg = SEM_UNDO | IPC_NOWAIT; /* take off semaphore,
  asynchronous  */

if ((j = semop(semid, sops, nsops)) == -1) {
   perror("semop: semop failed");
}
        else
  (void) fprintf(stderr, "Parent Process Giving up Control
   of Track: %d/3 times\n", i+1);
sleep(5); /* halt process to allow child to catch semaphor change
   first */
             }
   ++i;

}

     }
}
```

6 Mémoire partagée selon System V

On à vu qu'avec l'appel *fork*(), les variables n'étaient pas partagées entre les processus mais que chacun travaillait sur sa propre copie de celles-ci. Le seul moyen pour partager des données jusqu'ici était d'utiliser des *pipes* ou des *files* pour échanger des valeurs de variables. Tout ceci n'est bien évidement pas la solution la plus optimale car il y a là redondance des données. Heureusement, la norme System V définit des fonctions qui permettent de partager un espace mémoire (donc des données) entre plusieurs processus. Leur fonctionnement est des plus simple à l'instar des files de messages mais encore une fois, la version System V de la mémoire partagée n'est pas la plus efficace, l'implementation (POSIX) présente sur Solaris fonctionne mieux mais n'est malheureusement pas disponible sur Linux et *BSD.

La gestion de la mémoire partagée nécessite quelques fonctions que nous allons définir de suite.

La définition des structures et les prototypes des fonctions que nous allons utiliser se trouve dans les fichiers d'en-tête suivants : *sys/types. h*sys*/ipc.h* sys*/shm.*h

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

On utilise cette fonction pour obtenir l'accès à un segment de mémoire partagée.

key , tout comme pour les files de messages est un identifiant permettant à différents processus d'accéder à la même zone de mémoire partagée.

size est l'espace en octets alloué pour cette zone.

shmflg représente les *flags* de création et d'accès du segment. Le fonctionnement de ces flags est le même que celui des files.

La valeur de retour de cette fonction est un identifiant de segment de mémoire partagée. Si le a déjà été crée, la fonction renvoie jute l'identifiant.

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

Est utilisée pour changer les permissions du segment de mémoire partagée et d'autres options en fournissant les flags suivants dans l'argument *cmd* :

SHM_LOCK - Bloque le segment de mémoire. Le processus doit avoir EUID root pour effectuer cette action.

SHM_UNLOCK - Idem mais débloque le segment.

IPC_STAT - Retourne les informations de status contenues dans la structure de controle et les place dans le buffer pointé par *buf*.

IPC_SET - Change les permissions ainsi de l'EUID et le EGID. Le processus doit avoir l'EUID du propriétaire, du créateur ou du root pour utiliser cette commande.

IPC_RMID - Supprime le segment de mémoire partagée.

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

Permet d'attacher un segment de mémoire partagée précédemment crée avec *shmeget*() à un pointeur.

shmid est l'identifiant retourné par *shmget*().

shmaddr est l'adresse d'attachement.

shmflg représente les flags d'options pour l'attachement.

Si *shmaddr* vaut 0, le système va essayer de prendre lui même une zone mémoire libre dans l'intervalle 1-1.5G0.

Si *shmaddr* n'est pas nul et si *SHM_RND* est fournit dans *shmflg*, l'attachement l'attachement se fait l'adresse *shmaddr* arrondie au multiple inférieur de *SHMLBA*. Si *SHM_RND* n'est pas spécifié, *shmaddr* doit être aligné sur une frontière de page mémoire et l'attachement se fait à cet endroit.

Si *SHM_RDONLY* est spécifié, le segment est attaché en lecture seule. Sinon, le segment est attaché en lecture/écriture.

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

Détache le segment de mémoire partagé à l'adresse indiquée par par *shmaddr*.

Voici un exemple d'utilisation de cette implémentation :

Nous allons créer 2 programmes tout simples qui traitent une même chaîne (en mémoire partagée biensur). Le premier propose à l'utilisateur d'entrer une chaîne de caractères puis la place en mémoire partagée. Le seconde prend cette chaîne et la 'crypte'.

Premier programme :

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

#define TAILLE_MEMOIRE     64

int main(void)
{
   char c;
   int shmid;
   key_t key;
   char *shm, *s;
   key = 4242;

   if ((shmid = shmget(key, TAILLE_MEMOIRE, IPC_CREAT | 0666)) < 0) {
      perror("shmget");
      exit(-1);
```

```
   }

   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
      perror("shmat");
      exit(-1);
   }
   s = shm;
   printf("Entrez une phrase : ");
   fgets(s, TAILLE_MEMOIRE, stdin);
   c = shm[0];

   while (*shm == c)
     sleep(1);

   printf("\nVoici la chaine cryptée : %s\n",shm);

   if ((shmctl(shmid,IPC_RMID,(struct shmid_ds *) NULL))<0) {
      perror("shmctl");
      exit(-1);
   }

   return 0;
}
```

Second programme :

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

#define TAILLE_MEMOIRE     64

int main(void)
{
   int shmid;
   key_t key;
   char *shm;
   key = 4242;
   if ((shmid = shmget(key, TAILLE_MEMOIRE, 0666)) < 0) {
      perror("shmget");
      exit(-1);
   }
   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
      perror("shmat");
      exit(-1);
   }
   printf("Chaine trouvée : %s\n",shm);
   while (*shm != '\0') {
     *shm += 1;
      shm++;
   }
   printf("Chaine crypté.\n");
   return 0;
}
```

Dans ce cas (simple), seul un processus modifie la donnée partagée. Mais dans d'autres situations, il sera nécessaire d'établir des règles pour accéder à celle-ci dans un ordre précis. Les sémaphores ou les threads peuvent être bien utiles dans ce cas ...

7 Projection en mémoire façon BSD

La projection en mémoire est un moyen d'accéder à un fichier ou un périphérique de façon très efficace. C'est monstrueusement pratique ! Pour un fichier 'mappé' (projetté) en mémoire, la lecture et l'écriture des données se fait dans la RAM et non sur le disque, ce qui rend ces opérations beaucoup, beaucoup plus rapides . On peu aussi utiliser ce type de partage sur des périphériques tels qu'une carte son, une carte video, même une carte télé ... Grâce à cela, on va pouvoir accéder à la mémoire du périphérique lui-même. Pour réaliser cela nous avons besoin de ces fonctions :

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
```

Va allouer une zone de mémoire dans laquelle pourra être éventuellement projeté un fichier ou un périphérique.

start est l'emplacement mémoire ou l'on souhaite voir crée cette zone. Mais attention, ce n'est pas toujours cette adresse qui est prise !

lenght est le nombre d'octets qu'on va allouer.

prot indique les attributs de la zone de projection qu'on va utiliser. Ces options sont :

PROT_EXEC - On peut exécuter du code dans cette zone mémoire.

PROT_READ - On peut lire le contenu de cette zone.

PROT_WRITE - On peut écrire dans cette zone.

PROT_NONE - Le contenu de la zone est inaccessible. (- ? ?-)

flags indique le type de projection souhaité. Ce peut être :

MAP_FIXED - On n'utilise que l'adresse indiqué dans *start*. Si l'emplacement n'est pas disponible, la fonction échoue.

MAP_SHARED - La projection peut être partagée avec d'autres processus.

MAP_PRIVATE - La projection est privée, seul un processus père et ses fils peuvent y accéder. Les modifications n'affectent pas le fichier (ou périphérique) projeté.

MAP_ANONYMOUS - N'utilise pas de fichier en projection (Non POSIX).

fd est le déscripteur de fichier ou de périphérique à 'mapper' précédemment ouvert.

offset est un décalage par rapport au début du fichier ou du périphérique à partir duquel va commencer la projection.
Cette fonction renvoie un pointeur sur la zone mémoire si elle réussit, -1 avec *errno* mis à jour dans le cas contraire.

```
int munmap(void *start, size_t length);
```

Détruit la projection mémoire crée avec *mmap*().

La fonction renvoie 0 si elle réussit, -1 sinon.

```
int msync(cont void *start, size_t lenght, int flags);
```

Ecrit sur le disque (ou le périphérique) les modifications qui ont étés effectués dans la zone mémoire mappée.

Les structures et les prototypes des fonctions sont accessibles grâce aux entêtes *unistd.h* et *sys/mman.h*.

Nous allons maintenant voir grâce à *mmap* comment on peut effectuer des traitement sur un fichier sans nécessiter beaucoup d'accès disque :

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

#define TAILLE_MEMOIRE 4096

int main(void)
{
   int i=TAILLE_MEMOIRE-1;
   char *map,*c;
   int fd;

   if ((fd = open("/tmp/Zobby",O_CREAT | O_RDWR))<0) {
      perror("open");
      exit(-1);
   }
   if (ftruncate(fd,TAILLE_MEMOIRE+1)<0) {
      perror("ftruncate");
   }
   if ((c = map = mmap(0, TAILLE_MEMOIRE,
   PROT_WRITE, MAP_PRIVATE, fd,0))<0){
      perror("shmat");
      exit(-1);
   }

   /* Pour l'exemple nous effectuons une modification très simple
    * mais on pourrait imaginer un traitement plus lourd ...
    * dans ce cas, l'utilisation de projections s'avère plus efficace. */
   while (i--) {

      c[0] = 'b';
      c++;
   }

   if ((msync(map,TAILLE_MEMOIRE,MS_SYNC))<0) {
      perror("msync");
   }
   if ((munmap(map,TAILLE_MEMOIRE))<0) {
      perror("munmap");
   }
   close(fd);

   return 0;
}
```