LES I.P.C.

Licence de ce document

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	20 février 2005	Mise en ligne du document	© Frédéric Lang (frederic-lang@fr.fm)

SOMMAIRE

I) GENERALITES

1) Présentation

Les I.P.C. (Inter Process Communication) sont des outils permettant de faire communiquer deux processus d'une manière asynchrone, à l'inverse des tubes. C'est à dire qu'au moyen des I.P.C., un processus pourra avoir accès à une information alors que le processus qui a généré l'information est terminé depuis longtemps. La sauvegarde des informations est assurée par le système mais celui-ci ne garanti ces informations que s'il ne subit pas d'arrêt (pas de sauvegarde sur disque).
Les I.P.C. System V regroupent les objets suivants :

- ✓ les files de messages (xxx=msg)
- ✓ la mémoire partagée (xxx=shm). Cet objet offre une zone de mémoire accessible par tous les process
- ✓ les sémaphores (xxx=sem)

Leur mise en œuvre présente des points communs :

- ✓ les primitives de création et d'ouverture d'un objet se présentent sous la forme ***xxxget()***. Elles attendent une clef comme l'un des paramètres et renvoient un identificateur. La clef est une donnée de type ***key_t*** qui doit être connue de tous les processus utilisant l'I.P.C.
- ✓ les primitives de contrôle de l'objet se présentent sous la forme ***xxxctl()***. Elles permettent généralement d'obtenir les caractéristiques de l'objet, modifier ses caractéristiques ou détruire l'objet.
- ✓ chaque objet est géré par le noyau UNIX.

	Files de messages	**Mémoire partagée**	**Sémaphores**
Headers	sys/types.h sys/ipc.h sys/msg.h	sys/types.h sys/ipc.h sys/shm.h	sys/types.h sys/ipc.h sys/sem.h
Création / Ouverture	msgget()	shmget()	semget()
Contrôle	msgctl()	shmctl()	semctl()
Opérations	msgsnd() msgrcv()	shmat() shmdt()	semop()
Structures associées	msqid_ds	shmid_ds	semid_ds

Les I.P.C. utilisent les constantes prédéfinies suivantes définies dans <***sys/ipc.h***> :

- ✓ **IPC_CREAT** : création d'un nouvel élément (file de messages, zone mémoire, sémaphore)
- ✓ **IPC_EXCL** : échec de la création si élément existe déjà
- ✓ **IPC_ALLOC** : échec si l'élément n'existe pas déjà (n'existe pas sous *Linux* mais peut être remplacée par "***0***")
- ✓ **IPC_NOWAIT** : opération non bloquante
- ✓ **IPC_PRIVATE** : clef privée
- ✓ **IPC_RMID** : suppression d'un élément
- ✓ **IPC_STAT** : lecture du statut de la structure associée à l'élément
- ✓ **IPC_SET** : mise à jour de la structure associée à l'élément

2) Commandes shell de gestion des I.P.C.

La commande shell ***ipcs*** permet de consulter les tables I.P.C. du système. Sans option, cette commande permet de consulter toutes les tables. Ces renseignements sont :

- ✓ Type d'objet (q : file de message ; m : zone de mémoire partagée ; s : sémaphore)
- ✓ Identificateur de chaque objet
- ✓ Clef de chaque objet
- ✓ Mode (droits d'accès). En plus des indications habituelles, on trouve au premier caractère :
  - ☞ pour les files de message :
    - ☞ ***S*** si le processus est bloqué en écriture
    - ☞ ***R*** si le processus est bloqué en lecture
  - ☞ pour les zones de mémoire :
    - ☞ ***D*** si le segment est supprimé
    - ☞ ***C*** si le segment doit être initialisé lors du premier attachement
- ✓ Nom du propriétaire
- ✓ Nom du groupe
- ✓ Les options ***–q***, ***-m*** et ***–s*** permettent d'accéder aux tables particulières gérant chaque type d'objet (dans l'ordre : file de messages, zones de mémoire partagée, sémaphores).

Exemple :

```
$ ipcs
IPC status from /dev/kmem as of Mon Nov 15 15 :03 :17 1993
T       ID              KEY                 MODE                OWNER           GROUP
Message Queues :
q       0               0x0002fe03  Rrw-rw-rw-  root                    system
q       327681          0x00000017  -rw-rw-rw-          lambda          prof
q       131074          0x000001c8  -rw-rw-rw-          lambda          prof
Shared Memory :
m       0               0x00011e85  Crw-rw-rw-  root                    system
m       50              0x000001c9  -rw-rw-rw-          lambda          prof
Semaphores :
s       0               0x00011e85  -ra-ra-ra           root            system
s       1               0x00011e85  -ra-ra-ra-          root            system
```

La commande shell ***ipcrm*** permet de demander la suppression d'une entrée dans une des tables. L'entrée à supprimer peut être désignée soit par la clef associée (si elle n'est pas nulle, caractéristique d'une clef privé), soit par son identificateur interne.
Les arguments optionnels ***-q***, ***-m***, ***-s*** introduisent l'identificateur de l'objet du type correspondant alors que les arguments ***-Q***, ***-M***, ***-S*** introduisent une clef. Seul le propriétaire de l'objet peut demander sa suppression.

```
ipcrm -q ident        #suppression de la file d'identificateur ident
ipcrm -M clef         #suppression de la zone de mémoire clef
```

3) La structure "ipc_perm"

Chaque structure associée à un I.P.C. comprend, comme premier élément, une structure de type ***ipc_perm*** qui est définie dans **<*sys/ipc.h*>**.

```
#include <sys/types.h>
#include <sys/ipc.h>
struct ipc_perm {
   uid_t uid;          /* propriétaire de l'objet */
   gid_t gid;          /* groupe */
   uid_t cuid;         /* créateur */
   gid_t cgid;         /* groupe créateur */
   mode_t mode;        /* mode d'accès */
   u_long seq;         /* séquence numérique */
   key_t key;          /* clef de référence */
};
```

Explication des éléments :

- ✓ uid_t uid : numéro du propriétaire de l'objet
- ✓ gid_t gid : numéro de groupe de l'objet
- ✓ uid_t cuid : numéro du créateur de l'objet
- ✓ gid_t cgid : numéro de groupe de l'objet
- ✓ mode_t mode : droits de l'objet
- ✓ u_long seq : numéro quelconque
- ✓ key_t key : clef de l'objet

4) Création d'un canal I.P.C.

Les primitives ***xxxget()*** servent à créer un objet I.P.C. ("xxx" pouvant être "*msg*", "*shm*" ou "*sem*" selon que l'objet à créer sera une file de messages, une zone mémoire ou un ensemble de sémaphores).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/xxx.h>
int xxxget (key_t clef, [paramètres ou pas], int option);
```

Explication des paramètres :

✓ key_t clef : clef de l'objet

✓ [paramètres ou pas] : cela dépend en fait si la primitive ***msgget()***, ***shmget()*** ou ***semget()*** en a besoins. Ils seront alors spécifiques aux nécessités de la primitive.

✓ int option : option sur l'accès demandé. Celle-ci est composés des droits classiques octaux utilisés dans la primitive ***chmod()*** associés avec une ou plusieurs des quatre constantes suivantes :

- IPC_CREAT : création de l'objet s'il n'existe pas
- IPC_ALLOC : renvoie une erreur si l'objet n'existe pas (n'existe pas sous *Linux* mais peut être remplacée par "***0***")
- IPC_EXCL : envoie une erreur si l'objet existe déjà (protection)
- IPC_NOWAIT : renvoie une erreur s'il y a attente
- Autres constantes dépendantes spécifiquement de l'objet créé et dont l'explication dépend du chapitre traité

Valeur renvoyée (int) : descripteur de l'objet. Ce descripteur servira de référence ultérieure chaque fois qu'il faudra accéder à l'objet.

5) Création d'une clef

La clef est l'identificateur de l'objet. Cet identificateur doit être différent pour chaque objet distinct du système. Le système offre cependant plusieurs manières de créer une clef

✓ de manière directe : on utilise une valeur quelconque et arbitraire comme clef. Il est donc possible qu'un chiffre choisit "au hasard" par un programmeur entraîne l'utilisation d'un objet déjà existant (clef déjà choisie par un autre) ; et même ne soit pas portable entre plusieurs machines. C'est donc une politique à éviter

✓ avec une clef privée en utilisant la constante ***IPC_PRIVATE***. Cette constante utilisée comme clef garantit que l'objet créé le sera avec une clef qui n'est pas déjà utilisée. C'est pratique si l'objet est totalement manipulé par le programme qui l'a créé. Mais l'utilisation d'une clef privée ne permet pas à deux programmes distincts de manipuler un même objet.

✓ par l'utilisation de la primitive ***ftok()*** qui a pour but la génération de clefs uniques mais connues. L'utilisation de cette primitive par deux programmes distincts avec les mêmes paramètres conduiront à la génération de la même clef.

La primitive ***ftok()*** reçoit comme paramètre un nom de fichier et un numéro quelconque. Elle utilise l'inode correspondant au fichier et le numéro passés en paramètres afin de générer une clef. Le fichier doit exister et être accessible pendant la génération de la clef et le numéro ne doit pas être égal à ***0 modulo 256***.

```
#include <sys/types.h>
key_t ftok (char *nom, char ref);
```

Explication des paramètres :

- ✓ char *nom : nom du fichier pour générer la clef
- ✓ char ref : valeur arbitraire (sera tronquée sur un caractère qui ne doit pas être à égal à *0*).

Valeur renvoyée (key_t) : la clef générée.

La valeur de la clef est obtenue en récupérant le numéro d'inode associé au fichier et le numéro mineur du périphérique sur lequel réside le système de fichiers.
Ensuite, les 8 bits du caractère correspondant à la valeur "***ref***" tronquée sont simplement concaténés avec les 8 bits du numéro mineur du périphérique ; et avec les 16 bits de poids faibles du numéro d'inode pour générer un chiffre sur 32 bits.
L'algorithme ne garantit pas l'unicité de la clé. En fait :

- ✓ Deux noms différents pointant sur le même fichier (liens) produisent la même clé.
- ✓ L'utilisation des 16 bits de poids faibles du numéro d'inode laissent quelques (faibles) chances pour que deux inodes différents produisent la même clé.
- ✓ On n'utilise pas les numéros majeurs de périphériques, laissant encore quelques (faibles) possibilités de clef identiques sur un système avec plusieurs contrôleurs de disques qui auraient tous le même numéro mineur.

Exemple : Cet exemple permet la génération d'une clef différente selon l'endroit d'où est lancé le programme (il utilise comme fichier le répertoire courant "**.**") et selon celui qui lance le programme (il utilise comme référence le numéro d'utilisateur "***getuid()***"). Cela peut-être utilisé pour un projet où tous les sources du projet seront dans le même répertoire.

```
#include <sys/types.h>                          /* Types prédéfinis "c" */
#include <stdio.h>                              /* I/O bufferisées */
#include <errno.h>                              /* Gestion des erreurs */

extern const char const *sys_errlist[];         /* Liste des erreurs systèmes */

main()
{
        key_t clef;                             /* Clef de l'IPC à générer */

        /* Génération de la clef (le dernier octet du second argument ne doit pas être à 0) */
        if ((clef=ftok(".", (getuid() & 0xff) != 0 ?getuid() :-1)) == (-1))
                fprintf(stderr,"ligne %u – ftok() - %s\n", sys_errlist[errno]);

        /* Affichage clef générée */
        printf("Clef générée : 0x%04x\n", clef);
}
```

II) LES FILES DE MESSAGES

1) Généralités

Une file de messages est un objet manipulé par le système et identifié par un nombre entier (sa clef). Elle correspond au concept de boite aux lettres.
Un processus a la possibilité d'y déposer des messages ou d'en extraire. Un message est un couple formé d'un nombre entier (le type) et d'une suite d'octets de longueur arbitraire constituant le message proprement dit. Grâce au type, un processus récepteur pourra distinguer parmi les messages d'une file ceux qu'il souhaite extraire. Ce processus récepteur peut choisir de se mettre en attente soit sur le premier message disponible, soit sur le premier message d'un type donné. Un processus peut émettre des messages même si aucun processus n'est prêt à les recevoir. Les messages déposés sont conservés après la mort du processus émetteur, jusqu'à leur consommation ou la destruction de la file.

- ✓ les messages ne contiennent pas à priori le numéro du processus émetteur ni celui du destinataire
- ✓ c'est à l'utilisateur de décider de l'interprétation de chaque message
- ✓ une propriété essentielle de la file de messages est que le message forme un tout. On le dépose ou on l'extrait en une seule opération
- ✓ le récepteur peut choisir de lire soit le premier message globalement disponible, soit le premier message disponible d'un type donné

Pour que le mécanisme fonctionne, il faut qu'au préalable une file de messages ait été créée par un processus qui en sera le créateur propriétaire (la propriété pourra être transférée à un autre processus).

Les avantages principaux de la file de message par rapport aux tubes et aux tubes nommés sont :

- ✓ un processus peut émettre des messages même si aucun processus n'est prêt à les recevoir
- ✓ les messages déposés sont conservés, même après la mort de l'émetteur, jusqu'à leur consommation ou la destruction de la file.

Le principal inconvénient de ce mécanisme est la limite de la taille des messages.

Pour utiliser une file de message, on doit disposer des primitives suivantes :

- ✓ création et ouverture d'un file de message : ***msgget()***
- ✓ gestion ou contrôle de la file de messages : ***msgctl()***
- ✓ envoi ou réception des messages : ***msgsnd()*** et ***msgrcv()***

2) Eléments constitutifs d'une file de messages

Une file de messages est caractérisée par :

- ✓ une clef (définie par le créateur), qui pourra être utilisée par les autres processus pour accéder à la file
- ✓ un descripteur, retourné à la création, et utilisé pour les manipulations de la file au sein du processus
- ✓ lors de la création, une structure ***msqid_ds*** est créée, qui correspond à une entrée dans la table des files de messages gérées par le système. Cette structure identifie la file.

3) Les structures associées

a) La structure "msg"

La structure ***msg*** définie dans <***sys/msg.h***> contient les caractéristiques d'un message de la file. Cette structure est définie pour information mais n'est normalement pas accessible au programmeur.

```
#include <sys/msg.h>
struct msg {
   struct msg *msg_next;        /* pointeur sur le message suivant*/
   long msg_type;               /* type de message*/
   short msg_ts;                /* taille du texte du message*/
   char *msg_spot;              /* adresse du texte du message*/
};
```

Explication des éléments :

✓ struct msg *msg_next : pointeur sur une variable de type "***struct msg***" (structure identique) permettant le chaînage de la liste

✓ long msg_type : type du message

✓ short msg_ts : taille du corps du message

✓ char *msg_spot : pointeur sur une zone contenant le texte du message

b) La structure "msqid_ds"

La structure ***msqid_ds*** définie dans <***sys/msg.h***> contient les caractéristiques d'une file de messages.

```
#include <sys/msg.h>
struct msqid_ds {
   struct ipc_perm msg_perm;    /* permissions */
   struct msg *msg_first;       /* pointeur sur le premier message */
   struct msg *msg_last;        /* pointeur sur le dernier message */
   short msg_cbytes;            /* taille actuelle de la file */
   short msg_qnum;              /* nombre de messages de la file */
   short msg_qbytes;            /* taille totale de la file */
   short msg_lspid;             /* pid du dernier message envoyé */
   short msg_lrpid;             /* pid du premier message reçu */
   time_t msg_stime;            /* date du dernier message envoyé */
   time_t msg_rime;             /* date du dernier message reçu */
   time_t msg_ctime;            /* date du dernier changement */
};
```

Explication des éléments :

✓ struct ipc_perm msg_perm : variable de type ***struct ipc_perm*** caractérisant les permissions de la file de message

✓ struct msg *msg_first : pointeur sur une variable de type ***struct msg*** référençant le premier message de la file

✓ struct msg *msg_last : pointeur sur une variable de type ***struct msg*** référençant le dernier message de la file

✓ short msg_cbytes : taille actuelle (en octets) de la file de message

✓ short msg_qnum : nombre de messages dans la file

✓ short msg_qbytes : taille totale (en octets) de la file

✓ short msg_lspid : pid du dernier message envoyé

✓ short msg_lrpid : pid du dernier message reçu

✓ time_t msg_stime : date (en seconde depuis le 1/1/1970) du dernier message envoyé

✓ time_t msg_rtime : date (en seconde depuis le 1/1/1970) du dernier message reçu

✓ time_t msg_ctime : date (en seconde depuis le 1/1/1970) du dernier message changement de la file

c) La structure "msgbuf"

Un message, du point de vue du programmeur, est constitué par une structure générique de type ***struct msgbuf***.

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

Explication des éléments :

✓ mtyp_t mtype : type du message

✓ char mtext[1] : corps du message

Le terme "structure générique" signifie qu'elle est inutilisable telle qu'elle. Elle est seulement montrée à titre de squelette ou d'exemple indiquant au programmeur qu'il doit s'en créer une selon le schéma suivant :

✓ une variable de type ***mtyp_t*** représentant le type du message

✓ une zone de caractères contiguë en mémoire contenant le corps du message. Le terme "contiguë" signifie qu'il n'est pas autorisé d'y mettre un pointeur pointant vers une zone située en dehors de ladite structure.

4) Mise en œuvre d'une file de messages

a) Création ou utilisation d'une file de messages

La primitive ***msgget()*** sert à la création et/ou l'accès à une file de messages.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t clef, int option);
```

Explication des paramètres :

✓ key_t clef : clef de référence à la file de messages. La clef IPC_PRIVATE garantit l'unicité de la file de messages mais empêche l'accès à cette file par un autre processus.

✓ int option : option sur l'accès demandé. Celle-ci est composés des droits classiques octaux utilisés dans la primitive ***chmod()*** associés avec une ou plusieurs des six constantes suivantes :

- IPC_CREAT : création de la file de messages si elle n'existe pas
- IPC_ALLOC : renvoie une erreur si la file de messages n'existe pas (n'existe pas sous *Linux* mais peut être remplacée par "***0***")
- IPC_EXCL : renvoie une erreur si la file de messages existe déjà (protection)
- IPC_NOWAIT : renvoie une erreur s'il y a attente
- MSG_R : n'offre qu'une permission en lecture
- MSG_W : offre aussi une permission en écriture

Valeur renvoyée (int) : identifiant de la file de messages. Cet identifiant servira de référence ultérieure chaque fois qu'il faudra accéder à la file de messages.

b) Préparation d'un message

Il est nécessaire pour cela d'avoir définit une structure contenant dans l'ordre :

✓ un entier long (qui représentera le type du message). Ce type ne doit jamais être à zéro.

✓ une zone de caractères (qui représentera le corps du message)

Le programmeur n'a plus qu'à remplir les champs de sa structure avec les valeurs qu'il désire envoyer.

c) Envoi d'un message

La primitive ***msgsnd()*** permet d'envoyer un message dans la file de messages. Cette primitive recopie les octets de la structure définie par le programmeur dans la file de message. C'est pourquoi il n'est pas possible d'inclure un pointeur dans cette structure car l'objet pointé ne sera pas recopié.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msg_id, void *pt_msg, int size, int option);
```

Explication des paramètres :

✓ int msg_id : identifiant de la file de messages. Cet identifiant est retourné par la primitive ***msgget()***.

✓ void *pt_msg : pointeur universel (***void ****) vers une variable contenant le message à envoyer. Cette variable est en fait de type ***struct msgbuf*** créée et remplie par le programmeur.

✓ int size : taille du corps du message. Attention, ce paramètre ne doit pas prendre en compte la taille de l'entier prévu pour stocker le type de message.

✓ int option : option d'envoi du message. Celle-ci peut être :

- ☞ IPC_NOWAIT : l'appel à la fonction n'est pas bloquant. Si le message ne peut pas être envoyé (plus de place, etc.), la fonction retourne (***-1***) et la variable ***errno*** est positionnée à EAGAIN.
- ☞ 0 : l'appel est bloquant (standard). Tant que le message ne part pas, la fonction reste en attente.

Valeur renvoyée (int) : taille en octets du message déposé dans la file de messages.

d) Réception d'un message

La primitive ***msgrcv()*** permet de récupérer le premier message de la file de message correspondant au type voulu. Celui-ci, une fois récupéré, est enlevé de la file ; c'est à dire qu'aucun autre processus ne peut plus le récupérer, sauf si le premier récupérateur le redépose dans la file.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msg_id, void *pt_msg, int size, long type, int option);
```

Explication des paramètres :

✓ int msg_id : identifiant de la file de messages. Cet identifiant est retourné par la primitive ***msgget()***.

✓ void *pt_msg : pointeur universel (***void ****) vers une variable destinée à recevoir le message récupéré de la file. Cette variable est en fait de type ***struct msgbuf*** créée par le programmeur et remplie par la primitive ***msgrcv()***.

✓ int size : taille maximale pour le message à récupérer. La fonction renverra au plus le nombre d'octets réellement demandés. Mettre une taille à zéro est utile pour nettoyer une file.

✓ long type : correspond au type du message demandé. Cette valeur peut prendre trois cas différents :

- ☞ une valeur à ***0*** récupère le premier message de la file, sans tenir compte de leur type individuel.
- ☞ une valeur ***>0*** récupère le premier message correspondant au type demandé
- ☞ une valeur ***<0*** récupère le premier message dont le type est inférieur à la valeur absolue du nombre passé en paramètre

✓ int option : option d'envoi du message. Celle-ci peut être :

- ☞ IPC_NOWAIT : l'appel à la fonction n'est pas bloquant. Si le message ne peut pas être récupéré (plus de messages, etc.), la fonction retourne (***-1***) et la variable ***errno*** est positionnée à ENOMSG.
- ☞ 0 : l'appel est bloquant (standard). Tant qu'il n'y a pas de message à récupérer, la fonction reste en attente.

Valeur renvoyée (int) : taille en octets du message récupéré de la file de messages

e) Contrôle d'une file

Il est possible de réaliser certaines opérations de contrôle sur une file de messages grâce à la primitive ***msgctl()*** :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msg_id, int cmd, struct msqid_ds *buf);
```

Explication des paramètres :

✓ int msg_id : identifiant de la file de messages. Cet identifiant est retourné par la primitive ***msgget()***.

✓ int cmd : commande permettant de contrôler la file de message. Celle-ci doit être une des trois constantes suivantes :

- IPC_STAT : demande de statut sur la file de message
- IPC_SET : modification de statut sur la file de message
- IPC_RMID : effacement de la file de message

✓ struct msqid_ds *buf : pointeur vers une variable de type ***struct msqid_ds*** ; cette variable étant utilisée pour stocker tout le statut de la file de messages. Elle pourra servir pour obtenir une information (IPC_STAT), ou effectuer une mise à jour (IPC_SET) de la file de messages. Dans ce dernier cas, seuls les champs ***msg_perm.uid***, ***msg_perm.gid*** et ***msg_perm.mode*** peuvent être modifiés par l'utilisateur. Dans le cas d'un effacement (IPC_RMID), ce paramètre n'a aucune signification.

III) LES ZONES DE MEMOIRES PARTAGEES

1) Généralités

Le mécanisme de la mémoire partagée offre un espace d'adressage commun à plusieurs processus. Ce mécanisme permet à des processus distincts de partager une zone de données physique composée de "n" octets.
Sa mise en œuvre s'effectue ainsi :

- ✓ créer ou ouvrir une zone de mémoire partagée. A la création, on précise la taille (en octets) de la zone
- ✓ attacher la zone de mémoire partagée à un pointeur du programme (variable de type "*char**").
- ✓ utiliser la zone de mémoire partagée comme si c'était une zone de mémoire locale au programme
- ✓ gérer et contrôler la zone de mémoire partagée

En général, on combine l'utilisation de la mémoire partagée et des sémaphores (vus plus loin), de manière à gérer correctement l'accès à la mémoire partagée. En effet, alors que les accès aux données d'une file de message (vue précédemment) sont bloquants, donc atomiques, la mémoire partagée permet aux processus de manipuler directement ces données sans avoir à les recopier. Il faut donc faire attention aux accès simultanés sur une même zone. Aussi, pour assurer la cohérence des données lues, l'utilisation de celles-ci se fait généralement conjointement avec celle des sémaphores.

On remarquera, de plus, la différence de nature entre les mémoires partagées et les files de messages ; En effet, dans le cas de la mémoire partagée, les données lues par un processus sont conservées, même après la mort du processus émetteur ; jusqu'à leur modification par une écriture alors que les lectures sont destructives avec les files de message.

Pour utiliser une zone de mémoire partagée, on doit disposer des primitives suivantes :

- ✓ création et ouverture d'une zone de mémoire partagée : ***shmget()***
- ✓ gestion ou contrôle de la zone de mémoire partagée : ***shmctl()***
- ✓ attachement ou détachement d'une zone de mémoire partagée : ***shmat()*** et ***shmdt()***

2) Rappel des opérations possibles en zone mémoire

Etant donné que la zone de mémoire partagée est rattachée à un pointeur ("*char *pt*") du programme, il est très facile d'aller mettre n'importe quelle valeur comprise entre –128 et 127 (un octet) dans n'importe quelle case pointée par "pt".

```
pt[5]='a';          /* Mettre 'a' dans la 6ième case pointée par "pt" */
*(pt + 6)='b';      /* Mettre 'b' dans la 7ième case pointée par "pt" */
```

Cependant, lorsqu'il devient nécesaire de stocker un grand nombre d'octets dans la zone de mémoire partagée, ces opérations "octet par octet" peuvent devenir fastidieuses.
C'est pourquoi des fonctions de manipulations de zones mémoires ont été implémentées et sont d'une utilisation plus confortable pour effectuer des actions sur des groupes d'octets

a) Opérations sur zone mémoire

Les primitives ***memset()***, ***memcpy()*** et ***memcmp()*** permettent respectivement de remplir tous les octets d'une zone mémoire avec une valeur particulière, copier une zone mémoire dans une autre et comparer deux zones mémoires.

```
#include <sys/types.h>
void *memset (void *buffer, int val, size_t nb);
void *memcpy (void *dest, void *source, size_t nb);
int memcmp (void *buf1, void *buf2, size_t nb);
```

Explication des paramètres :

- ✓ void *buffer : pointeur universel (***void ****) vers la zone mémoire que l'on désire initialiser
- ✓ void *dest : pointeur universel (***void ****) vers la zone mémoire destinatrice de la recopie
- ✓ void *source : pointeur universel (***void ****) vers la zone mémoire contenant la source à recopier
- ✓ void *buf1 : pointeur universel (***void ****) vers la zone mémoire n° 1 que l'on désire comparer
- ✓ void *buf2 : pointeur universel (***void ****) vers la zone mémoire n° 2 que l'on désire comparer
- ✓ int val : valeur d'initialisation
- ✓ size_t nb : nombre d'octets sur lesquels s'appliquera chaque fonction

Explication des paramètres (int) : résultat de la comparaison.

- ✓ 0 : chaque caractère des deux zones est identique
- ✓ n<0 : la zone 1 est inférieure à la zone 2 au caractère ***-n***
- ✓ n>0 : la zone 1 est supérieure à la zone 2 au caractère ***n***

b) Opérations sur chaînes de caractères

Ces fonctions sont d'un emploi plus aisé que les fonctions précédentes, où il faut sans cesse indiquer le nombre d'octets sur lesquels elles s'appliquent ; mais ont comme impératif les trois règles suivantes :

✓ Chaque tableau de caractères qui est passé à une fonction pour être lu doit contenir quelque part la valeur **'\0'** permettant à la fonction de repérer la fin du tableau

✓ Chaque tableau de caractères qui est passé à une fonction pour y être écrit contiendra au sortir de la fonction la valeur **'\0'** écrite par cette dernière et doit donc être de taille suffisante pour stocker cette valeur supplémentaire

✓ Chaque pointeur éventuellement renvoyé par la fonction pointera sur un tableau de caractères contenant lui-aussi une valeur **'\0'** positionnée par la fonction

Ces règles sont la base des techniques de manipulation de chaînes de caractères en langageC.

La fonction ***strlen()*** permet de compter la longueur d'une chaîne de caractères. La fonction cherche et calcule la position du *'\0'* dans le tableau de caractères reçu en argument et renvoie cette position (d'où l'utilité que le programmeur soit certain que ce *'\0'* soit bien présent dans le tableau envoyé à la fonction).

Il est important de faire la distinction entre l'opérateur ***sizeof()*** qui renvoie la taille globale d'une zone mémoire et la fonction ***strlen()*** qui indique combien sont présents de caractères avant le *'\0'*.

```
#include <string.h>
int strlen (char *chaine);
```

Explication des paramètres :

✓ char *chaine : La chaîne dont on désire connaître la longueur

Valeur renvoyée (int) : La longueur de la chaîne

Les fonctions ***strcmp()*** et ***strncmp()*** permettent de comparer deux chaînes de caractères. La comparaison se fera octet par octet jusqu'à ce qu'il y ait une différence et la fonction ***strncmp()*** ne fera la comparaison que sur les "n" premiers octets.

```
#include <string.h>
int strcmp (char *ch1, char *ch2);
int strncmp (char *ch1, char *ch2, size_t n);
```

Explication des paramètres :

✓ char *ch1 et char *ch2 : Les deux chaînes à comparer

✓ size_t n : le nombre de caractères sur lesquels se fera la comparaison

Valeur renvoyée (int) : Ces fonctions renvoient trois types de valeur :

✓ Une valeur inférieure à zéro si "ch1" est plus petit que "ch2"

✓ Une valeur égale à zéro si "ch1" est égal à "ch2"

✓ Une valeur supérieure à zéro si "ch1" est plus grand que "ch2"

Les fonctions ***strcpy()***, ***strncpy()***, ***strcat()*** et ***strncat()*** permettent respectivement de copier une chaîne dans une autre, copier seulement les "n" premiers caractères d'une chaîne dans une autre, concaténer une chaîne à la suite d'une autre et de concaténer seulement les "n" premiers caractères d'une chaîne à la suite d'une autre.

```
#include <string.h>
char *strcpy (char *dest, char *src);
char *strncpy (char *dest, char *src, size_t n);
char *strcat (char *dest, char *src);
char *strncat (char *dest, char *src, size_t n);
```

Explication des paramètres :

- ✓ char *dest : La chaîne de destination (qui doit avoir la place de recevoir ce que la fonction lui mettra plus le caractère '\0')
- ✓ char *src : La chaîne source
- ✓ size_t n : le nombre de caractères pris de la chaîne source

Valeur renvoyée (char*) : Ces fonctions renvoient un pointeur sur la chaîne de destination.

Les fonctions ***strchr()***, ***strrchr()*** et ***strstr()*** permettent respectivement de recherche un caractère dans une chaîne en partant du début ou en partant de la fin de la chaîne ; et de rechercher une sous-chaîne contenue dans une chaîne.

```
#include <string.h>
char *strchr (char *chaîne, int carac);
char *strrchr (char *chaîne, int carac);
char *strstr (char *chaîne, char *sousch);
```

Explication des paramètres :

- ✓ char *chaîne : La chaîne de recherche
- ✓ int carac : Le caractère à rechercher (converti en entier mais cela n'a pas d'importance puisque l'entier a un codage plus grand que le caractère)
- ✓ char *sousch : La sous-chaîne à rechercher

Valeur renvoyée (char*) : Ces fonctions renvoient un pointeur sur le caractère trouvé ou un pointeur sur le début de la sous-chaîne trouvée.

3) Eléments constitutifs d'une zone de mémoire partagée

Une zone de mémoire partagée est caractérisée par :

✓ une clef (définie par le créateur), qui pourra être utilisée par les autres processus pour accéder à la zone

✓ un descripteur, retourné à la création, et utilisé pour les manipulations de la zone de mémoire partagée au sein du processus

✓ lors de la création, une structure ***shmid_ds*** est créée, qui correspond à la zone de mémoire partagée gérée par le système. Cette structure identifie la zone de mémoire partagée.

4) Les structures associées

a) La structure "shminfo"

La structure ***shminfo*** définie dans **<*sys/shm.h***> donne les informations sur l'ensemble des segments définis dans le système. Cette structure est donnée pour information mais n'est normalement pas accessible.

```
#include <sys/shm.h>
struct shminfo {
   int shmmax;              /* taille max d'un segment */
   int shmmin;              /* taille min d'un segment */
   int shmmni;              /* nombre d'identificateurs */
   int shmseg;              /* nombre max segments par processus */
   int shmall;              /* nombre max segments pour système */
};
```

Explication des éléments :

✓ int shmmax : taille maximale d'un segment

✓ int shmmin : taille minimale d'un segment

✓ int shmmni : nombre d'identificateurs

✓ int shmseg : nombre maximal de segments par processus

✓ int shmall : nombre maximal de segments pour le système

b) La structure "shmid_ds"

La structure ***shmid_ds*** définie dans <***sys/shm.h***> contient les caractéristiques d'une zone de mémoire partagée.

```
#include <sys/shm.h>
struct shmid_ds {
   struct ipc_perm shm_perm;    /* permissions */
   int shm_segsz;               /* taille du segment de mémoire */
   long *shm_paddr;             /* adresse physique du segment */
   short shm_lpid;              /* pid du dernier processus utilisateur */
   short shm_cpid;              /* pid du processus createur */
   short shm_nattch;            /* nombre de processus attachés */
   short shm_cnattch;           /* nombre de processus en mémoire */
   time_t shm_atime;            /* date du dernier attachement */
   time_t shm_dtime;            /* date du dernier detachement */
   time_t shm_ctime;            /* date du dernier changement */
};
```

Explication des éléments :

✓ struct ipc_perm shm_perm : variable de type ***struct ipc_perm*** caractérisant les permissions de la zone de mémoire partagée

✓ int shm_segz : taille de la zone de mémoire partagée

✓ long *shm_paddr : pointeur sur une variable de type long référençant l'adresse du segment dans la mémoire du système

✓ short shm_lpid : pid du dernier processus utilisateur

✓ short shm_cpid : pid du processus créateur

✓ short shm_nattach : nombre de processus attachés

✓ short shm_cnattach : nombre de processus en mémoire

✓ time_t shm_atime : date (en seconde depuis le 1/1/1970) du dernier attachement de la zone

✓ time_t shm_dtime : date (en seconde depuis le 1/1/1970) du dernier détachement de la zone

✓ time_t shm_ctime : date (en seconde depuis le 1/1/1970) du dernier message changement de la zone

5) Mise en œuvre d'une zone de mémoire partagée

a) Création ou utilisation d'une zone de mémoire partagée

La primitive ***shmget()*** sert à la création et/ou l'accès à une zone de mémoire partagée.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t clef, int size, int options);
```

Explication des paramètres :

✓ key_t clef : clef de référence à la zone de mémoire partagée. La clef IPC_PRIVATE garantit l'unicité de la zone mais empêche l'accès à cette zone pour un autre processus.

✓ int size : taille de la zone demandée. Celle-ci doit être compatible avec les limites imposées par le système ou avec la taille de la zone si elle existe déjà.

✓ int option : option sur l'accès demandé. Celle-ci est composés des droits classiques octaux utilisés dans la primitive ***chmod()*** associés avec une ou plusieurs des six constantes suivantes :

- IPC_CREAT : création de la zone si elle n'existe pas
- IPC_ALLOC : renvoie une erreur si la zone n'existe pas (n'existe pas sous *Linux* mais peut être remplacée par "***0***")
- IPC_EXCL : ne pas accéder à la zone si elle existe déjà (protection)
- IPC_NOWAIT : renvoie une erreur s'il y a attente
- SHM_R : n'offre qu'une permission en lecture
- SHM_W : offre aussi une permission en écriture (par défaut)

Valeur renvoyée (int) : identifiant de la zone de mémoire partagée. Cet identifiant servira de référence ultérieure chaque fois qu'il faudra accéder à la zone de mémoire partagée.

b) Attachement d'une zone de mémoire partagée

La primitive ***shmat()*** permet d'attribuer à un processus une adresse virtuelle correspondant à une adresse physique de la zone de mémoire partagée. Ce n'est qu'alors qu'il peut accéder à la zone. Cette opération se nomme "attachement".

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void *shmat (int shm_id, char *pt_shm, int option);
```

Explication des paramètres :

✓ int shm_id : identifiant de la zone de mémoire partagée. Cet identifiant est retourné par la primitive ***shmget()***.

✓ char *pt_shm : pointeur sur une zone mémoire du processus à laquelle le segment de mémoire partagé sera attaché. Le problème qui se pose est que le programmeur ne sait pas forcément où attacher. Cela ne doit pas être une adresse déjà utilisée, ni empêcher l'augmentation de la taille de la zone de données ni celle de la pile ; elle doit être compatible avec les formes d'adresses gérées par le système. Afin d'assurer la portabilité des applications, il est recommandé de s'en remettre au système pour cette opération en utilisant la valeur ***zéro*** pour ce paramètre. Il est également possible de choisir l'adresse avec plus ou moins de liberté selon la taille du segment. Il y a alors deux possibilités :

- soit l'utilisateur fixe une adresse absolue et le système essaie de rattacher le segment à cette adresse
- soit il fournit une adresse relative par rapport à un début de page de donnée et le système se chargera de la transformer en une adresse absolue. il faut alors positionner le bit SHM_RND dans ***option***.

✓ int option : option d'attachement de la zone. Celle-ci peut être :

- 0 : pas d'option spécifique
- SHM_DEST : destruction du segment quand aucun processus n'est plus attaché
- SHM_RONLY : attachement en lecture seule
- SHM_RND : spécifie que l'adresse passée à ***pt_shm*** est relative par rapport à un début de page (cf. mémoire paginée ou swapée). Le système se chargera alors de transformer cette adresse relative en adresse absolue.

Valeur renvoyée (void *) : adresse universelle de la zone de mémoire partagée à laquelle peut maintenant accéder le processus.

c) Détachement d'une zone de mémoire partagée

La primitive ***shmdt()*** permet de détacher d'un processus une zone de mémoire partagée sur laquelle il ne désire plus avoir accès. Le segment détaché n'est réellement libéré que lorsque le nombre d'attachements pointants dessus devient nul.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt (char *pt_shm);
```

Explication des paramètres :

✓ void *pt_shm : pointeur sur la zone mémoire du processus à détacher

d) Contrôle d'une zone de mémoire partagée

Il est possible de réaliser certaines opérations de contrôle sur une zone de mémoire partagée grâce à la primitive ***shmctl()*** :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shm_id, int cmd, struct shmid_ds *buf);
```

Explication des paramètres :

✓ int shm_id : identifiant de la zone de mémoire partagée. Cet identifiant est retourné par la primitive ***shmget()***.

✓ int cmd : commande permettant de contrôler la zone de mémoire partagée. Celle-ci doit être une des cinq constantes suivantes :

- IPC_STAT : demande de statut sur la zone de mémoire partagée
- IPC_SET : modification de statut sur la zone de mémoire partagée
- IPC_RMID : effacement de zone de mémoire partagée. Dans ce cas, la commande empêche tout attachement du segment par un nouveau processus et ne supprime le segment que lors du dernier détachement.
- SHM_LOCK : retire de la pagination le segment de mémoire partagé. Le segment devient verrouillé. Cette commande n'est autorisée qu'à un processus dont l'utilisateur effectif (***euid***) est 0 (***super utilisateur*** ou ***root***).
- SHM_UNLOCK : remet dans la pagination le segment de mémoire partagé. Le segment devient accessible. Cette commande n'est autorisée qu'à un processus dont l'utilisateur effectif (***euid***) est 0 (***super utilisateur*** ou ***root***).

✓ struct shmid_ds *buf : pointeur vers une variable de type ***struct shmid_ds*** ; cette variable étant utilisée pour stocker tout le statut de la zone de mémoire partagée. Elle pourra servir pour obtenir une information (IPC_STAT), ou effectuer une mise à jour (IPC_SET) de la zone de mémoire partagée. Dans ce dernier cas, seuls les champs ***shm_perm.uid***, ***shm_perm.gid*** et ***shm_perm.mode*** peuvent être modifiés par l'utilisateur. Dans le cas d'un effacement (IPC_RMID), d'un verrouillage (SHM_LOCK) ou d'un déverrouillage (SHM_UNLOCK), ce paramètre n'a aucune signification.

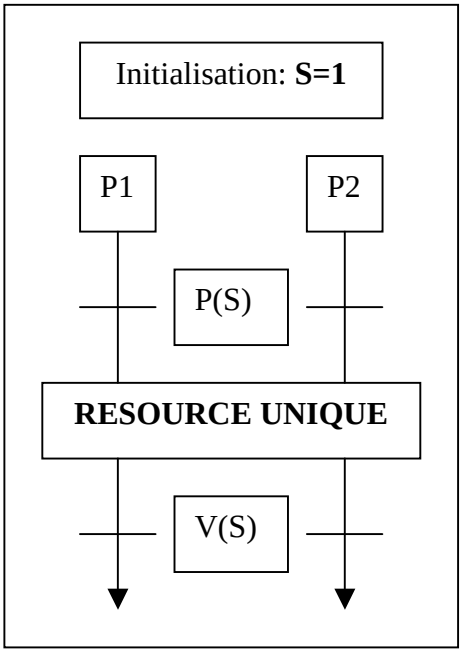
Il est bien évident que si deux ou plusieurs processus écrivent et lisent "en même temps" dans la zone de mémoire partagée, il n'est pas possible de savoir ce qui est réellement pris en compte. D'où l'obligation d'utiliser des sémaphores pour ne donner l'accès à cette zone qu'à un processus à la fois.

IV) LES SEMAPHORES

1) Généralités

Les sémaphores constituent avant tout un mécanisme de synchronisation des processus. Un sémaphore S est une variable à valeurs entières non négatives accessibles à partir de deux opérations particulières appelées aussi ***opérations atomiques de Dijkstra*** :

- ✓ P(S) :
  - ☞ si S=0 alors mettre le processus en attente
  - ☞ sinon S=S-1 et laisser passer le processus
- ✓ V(S) :
  - ☞ S=S+1
  - ☞ réveiller un processus en attente



Les sémaphores sous UNIX s'utilisent de la manière suivante :

- ✓ créer ou ouvrir un ou plusieurs sémaphores grâce à la primitive ***semget()***
- ✓ initialiser les sémaphores
- ✓ incrémenter ou décrémenter les compteurs des sémaphores avec la primitive ***semop()***
- ✓ gérer et contrôler ces sémaphores avec la primitive ***semctl()***

2) Eléments constitutifs d'un ensemble de sémaphores

Un ensemble de sémaphores est caractérisé par :

✓ une clef (définie par le créateur), qui pourra être utilisée par les autres processus pour accéder à l'ensemble

✓ un descripteur, retourné à la création, et utilisé pour les manipulations de l'ensemble au sein du processus

✓ lors de la création, une structure ***semid_ds*** est créée, qui correspond à une entrée dans la table des ensemble de sémaphores gérés par le système. Cette structure identifie l'ensemble

3) Les structures associées

a) La structure "__sem"

La structure ***__sem*** définie dans ***<sys/sem.h*** correspond à la structure dans le noyau d'un sémaphore individuel. Cette structure est définie pour information mais n'est normalement pas accessible.

```
#include <sys/sem.h>
struct __sem {
   unsigned short int semval;     /* valeur du sémaphore */
   unsigned short int sempid;     /* pid du dernier processus utilisateur */
   unsigned short int semncnt;    /* nb attentes augmentation sémaphore */
   unsigned short int semzcnt;    /* nombre attentes nullité sémaphore */
};
```

Explication des éléments :

- ✓ unsigned short int semval : valeur numérique du sémaphore
- ✓ unsigned short int sempid : pid du dernier processus utilisateur
- ✓ unsigned short int semncnt : nombre de processus attendant l'augmentation du sémaphore
- ✓ unsigned short int semzcnt : nombre de processus attendant la nullité du sémaphore

b) La structure "semid_ds"

La structure ***semid_ds*** définie dans ***<sys/sem.h***> contient les caractéristiques d'un ensemble de sémaphores.

```
#include <sys/sem.h>
struct semid_ds {
   struct ipc_perm sem_perm;     /* permissions */
   struct __sem *sem_base;       /* pointeur premier sémaphore */
   unsigned short sem_nsems;     /* nombre sémaphores */
   time_t sem_otime;             /* date de dernière opération */
   time_t sem_ctime;             /* date du dernier changement */
   };
```

Explication des éléments :

✓ struct ipc_perm sem_perm : variable de type ***struct ipc_perm*** caractérisant les permissions de l'ensemble de sémaphores

✓ struct __sem *sem_base : pointeur sur une variable de type ***struct __sem*** référençant l'adresse du tableau des sémaphores de l'ensemble

✓ unsigned short sem_nsems : nombre de sémaphores de l'ensemble

✓ time_t sem_otime : date (en seconde depuis le 1/1/1970) de la dernière opération sur un des éléments de l'ensemble

✓ time_t sem_ctime : date (en seconde depuis le 1/1/1970) du dernier changement de l'ensemble

c) La structure "sembuf"

Les actions sur un sémaphore (augmentation ou diminution) sont décrites dans une structure de type ***struct sembuf*** définie dans <***sys/sem.h***>

```
#include <sys/sem.h>
struct sembuf {
   unsigned short sem_num;      /* numéro du sémaphore */
   short sem_op;                /* opération sur le sémaphore */
   short sem_flg;               /* flag = 0 */
};
```

Explication des éléments :

✓ unsigned short sem_num : numéro du sémaphore à opérer. Ce numéro commence à ***0*** pour le premier sémaphore.

✓ short sem_op : opération demandée. Ce paramètre peut prendre trois valeurs :
- -n => demande de décrément de *n* (qui correspond à ***P(S)***) si cette opération est possible. Si elle ne l'est pas, le processus est mis en attente que cette opération soit possible.
- +n => demande d'incrément de *n* (qui correspond à ***V(S)***). Une fois que cette opération est exécutée, un autre processus en attente de *P(S)* peut exécuter cette opération.
- 0 => attente que le sémaphore devienne nul (opération ***Z(S)***).

✓ short sem_flg : flag d'opération. Ce paramètre peut prendre une des trois valeurs suivantes :
- 0 => opération normale
- SEM_UNDO => l'opération sera annulée lorsque le process qui l'a demandée se terminera. Cette option est utilisée pour être sûr qu'un programme ne se finit pas sans remettre le sémaphore dans l'état original.
- IPC_NOWAIT => opération non bloquante. Il peut parfois être nécessaire de ne pas bloquer le process même si le sémaphore ne peut pas être pris. Dans ce cas, la fonction retourne (***-1***) et la variable ***errno*** est positionnée à EAGAIN.

4) Mise en œuvre d'un ensemble de sémaphores

a) Création ou utilisation d'un ensemble de sémaphores

La primitive ***semget()*** sert à la création et/ou l'accès à un ensemble de sémaphores.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t clef, int nb_sem, int option);
```

Explication des paramètres :

✓ key_t clef : clef de référence à l'ensemble de sémaphores. La clef IPC_PRIVATE garantit l'unicité de l'ensemble mais empêche l'accès à cet ensemble par un autre processus.

✓ int nb_sem : nombre de sémaphores de l'ensemble (limité à 64)

✓ int option : option sur l'accès demandé. Celle-ci est composés des droits classiques octaux utilisés dans la primitive ***chmod()*** associés avec une ou plusieurs des six constantes suivantes :

- IPC_CREAT : création de l'ensemble de sémaphores si il n'existe pas
- IPC_ALLOC : renvoie une erreur si l'ensemble de sémaphores n'existe pas (n'existe pas sous *Linux* mais peut être remplacée par "***0***")
- IPC_EXCL : renvoie une erreur si l'ensemble de sémaphores existe déjà (protection)
- IPC_NOWAIT : renvoie une erreur s'il y a attente
- SEM_R : n'offre qu'une permission en lecture
- SEM_A : offre aussi une permission d'altération (incrémentation/décrémentation) mise par défaut

Valeur renvoyée (int) : identifiant de l'ensemble de sémaphores. Cet identifiant servira de référence ultérieure chaque fois qu'il faudra accéder à l'ensemble.

b) Préparation d'une opération

Il est possible de regrouper dans un "tout" plusieurs opérations atomiques de ***Dijkstra***, chacune de ces opérations s'effectuant sur un sémaphore précis de l'ensemble de sémaphores. Il suffit de définir un tableau de type ***struct sembuf*** . Chaque élément du tableau sera rempli par le programmeur afin qu'il contienne l'opération à effectuer et le sémaphore sur lequel opérer. Ce groupe d'opérations sera atomique, c'est à dire qu'il sera exécuté sur tous les sémaphores du groupe ou sur aucun.
Si l'opération ne concerne qu'un seul sémaphore, une variable suffit au lieu d'un tableau.

c) Opération de Dijkstra

La primitive ***semop()*** permet de demander un groupe d'opérations de ***Dijkstra*** sur un groupe de sémaphores de l'ensemble. Ce groupe d'opérations sera atomique, c'est à dire qu'il sera exécuté sur tous les sémaphores du groupe ou sur aucun.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop (int sem_id, struct sembuf *pt_op, unsigned int nb_op);
```

Explication des paramètres :

✓ int sem_id : identifiant de l'ensemble de sémaphores. Cet identifiant est retourné par la primitive ***semget()***.

✓ struct sembuf *pt_op : pointeur sur une variable de type ***struct sembuf*** référençant l'adresse du tableau des opérations à effectuer

✓ unsigned int nb_op : nombre d'opérations à effectuer (correspond au nombre d'éléments du tableau)

d) Contrôle d'un ensemble de sémaphores

Il est possible de réaliser certaines opérations de contrôle sur un ensemble de sémaphores grâce à la primitive ***semctl()*** :

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int sem_id, int semnum, int cmd, union semun {
        int val;
        struct semid_ds *buf;
        unsigned short *tab
} arg);
```

Explication des paramètres :

✓ int sem_id : identifiant de l'ensemble de sémaphores. Cet identifiant est retourné par la primitive ***semget()***.

✓ int semnum : numéro du sémaphore sur lequel il faut réaliser une opération. Dans le cas où l'opération concerne l'ensemble de tous les sémaphores et non un sémaphore particulier, ce paramètre peut-être d'une valeur quelconque car celle-ci ne sera pas utilisée ; mais il faut y mettre une valeur pour conserver l'ensemble des paramètres.

✓ int cmd : commande permettant de contrôler l'ensemble de sémaphores. Celle-ci doit être une des dix constantes suivantes :

- IPC_STAT : demande de statut sur l'ensemble de sémaphores
- IPC_SET : modification de statut de l'ensemble de sémaphores
- IPC_RMID : effacement de l'ensemble de sémaphores
- GETNCNT : interrogation sur le nombre de processus en attente d'augmentation du sémaphore référencé par ***semnum***
- GETZCNT : interrogation sur le nombre de processus en attente de nullité du sémaphore référencé par ***semnum***
- GETPID : interrogation sur le dernier processus ayant réalisé une opération sur le sémaphore référencé par ***semnum***
- GETVAL : interrogation sur la valeur du sémaphore référencé par ***semnum***
- GETALL : interrogation sur la valeur de tous les sémaphores de l'ensemble
- SETVAL : affectation d'une valeur précise dans le sémaphore référencé par ***semnum***
- SETALL : affectation d'une valeur précise dans tous les sémaphores de l'ensemble

✓ union semun {…} arg : argument pouvant être indifféremment considéré comme :

- int val : valeur d'affectation pour un sémaphore (SETVAL)
- struct semid_ds *buf : pointeur vers une variable de type ***struct semid_ds*** ; cette variable étant utilisée pour stocker tout le statut de l'ensemble de sémaphores. Elle pourra servir pour obtenir une information (IPC_STAT), ou effectuer une mise à jour (IPC_SET) de l'ensemble des sémaphores. Dans ce dernier cas, seuls les champs ***sem_perm.uid***, ***sem_perm.gid*** et ***sem_perm.mode*** peuvent être modifiés par l'utilisateur.
- unsigned short *tab : pointeur sur une variable de type ***unsigned short*** référençant l'adresse du tableau des valeurs des sémaphores. Ce tableau peut-être destiné à recevoir les valeurs (GETALL) ou à affecter des valeurs (SETALL).
- sans signification précise pour le cas des commandes GETNCNT, GETZCNT, GETPID, GETVAL et IPC_RMID

Valeur renvoyée (int) : cette valeur a des significations différentes suivant la valeur du paramètre ***cmd*** qu'a reçu la fonction :

✓ GETNCNT : nombre de processus en attente d'augmentation du sémaphore référencé par ***semnum***

✓ GETZCNT : nombre de processus en attente de nullité du sémaphore référencé par ***semnum***

✓ GETPID : numéro du dernier processus ayant réalisé une opération sur le sémaphore référencé par ***semnum***

✓ GETVAL : valeur du sémaphore référencé par ***semnum***

INDEX