Cours "Système d'exploitation"

R3.05

2ème année BUT

Département d'informatique Caen

*Développement système : **les IPC***

Table des matières

I/ Introduction : 3
I-1/ Gestion des clés 4
I-2/ Commande shell 4
II/Les mémoires partagées 5
II-1/ Création de mémoire partagée 5
II-2/ attachement/détachement à un segment de mémoire partagée 5
II-3/ Contrôle d'un segment de mémoire partagée 6
II-4/Exemples 7
Exemple1 : CreerZDC.c 7
Exemple2 : LireZDC.c 9
Exemple3 : DetruireZDC.c 11
III/Les sémaphores 12
III-1/Introduction : 12
III-2/Création de sémaphore : 14
III-3/Prendre/vendre un sémaphore : 15
III-4/Contrôle de sémaphore : 16
III-5/Rendez-vous de processus : opération Z : 18
IV/ les queues (ou files) de messages : 19
IV-1/Introduction 19
IV-2/Création d'une file de messages 20
IV-3/Lecture d'un message dans une file de message 21
IV-4/ Ecriture d'un message dans une file de message 21
IV-5/Contrôle d'une file de messages 21
IV-6/Exemples 22
Exemple1 : Création d'une file de messages 23
Exemple2 : lecture d'un message dans la file de messages 24
Exemple3 : Informations sur la file de message 24

Partie 4 : les IPC

I/ Introduction :

Les IPC (***InterProcess Communication***) regroupent un ensemble de mécanismes permettant à des processus distincts de communiquer.

Ce sont des mécanismes de communication qui permettent de manipuler des données de tout type, de les envoyer d'un processus à un autre ou bien de les partager entre processus. Ils sont au nombre de trois :

- ✓ Les mémoires partagées : permettent de partager une zone de mémoire entre plusieurs applications,
- ✓ Les sémaphores : permettent de synchroniser les processus lorsqu'ils accèdent aux mêmes données par exemple,
- ✓ Les files de messages : permettent à une application, sous réserve qu'elle en ait les droits nécessaires, d'y déposer un message ou d'y lire un message.

Ces trois mécanismes sont externes au système de gestion de fichiers et on ne pourra donc pas utiliser les fonctions d'ouverture, de fermeture, de lecture ou d'écriture habituelles (open, close, read et write). De manière externe, ils ne seront donc pas référencés dans le système de fichiers mais identifiés par un mécanisme de clés. Le système assure la gestion de chacun des trois types de mécanismes et est ainsi capable de les distinguer.

Les IPC permettent de faire communiquer deux processus d'une manière asynchrone, à l'inverse des tubes. C'est à dire qu'au moyen des IPC, 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 garantit ces informations que s'il ne subit pas d'arrêt (car il n'y a pas de sauvegarde sur disque dans ce cas). Dans tous les cas, ces outils de communication peuvent être partagés entre des processus n'ayant pas immédiatement d'ancêtre commun.

La mise en œuvre des IPC 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'IPC.
- ✓ 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.

Tout objet, géré par le noyau, est identifié par :

- ✓ Un identificateur interne correspondant à l'entrée dans une table système et contenant les caractéristiques de l'objet,
- ✓ Un identificateur externe, appelé la clé, utilisé par les processus utilisateurs pour manipuler l'objet

I-1/ Gestion des clés

Afin d'identifier de manière unique une file de message, une mémoire partagée ou un sémaphore, une clé est utilisée. Elle peut être créée par la fonction

key_t ftok (char *pathname, int proj)

qui à partir du numéro d'inode (unique) du fichier ***pathname*** et de l'identificateur ***proj*** crée un numéro unique. Cette clé est propre au mécanisme et un même numéro peut être la clé d'une file de message, d'une mémoire partagée ou d'un sémaphore.

I-2/ Commande shell

La commande ***ipcs*** permet de consulter les tables IPC du système. La commande ***ipcrm*** permet de détruire une ressource IPC.

Exemple :

```
nouzhatber@ServeurLinux:~/ExCours/QueueDeMessages$ ipcs

------ Files de messages --------
clef       msqid      propriétaire perms      octets utilisés messages

------ Segment de mémoire partagée --------
clef       shmid      propriétaire perms      octets     nattch     états

------ Tableaux de sémaphores --------
clef       semid      propriétaire perms      nsems
```

La commande ***ipcs*** donne un certain nombre d'informations :

- ✓ **clef** : correspond à la clé de création de l'IPC,
- ✓ **msqid, shmid** ou **semid** est l'identificateur de l'IPC,
- ✓ **propriétaire** est comme son nom l'indique, le propriétaire de l'IPC,
- ✓ **perms** représente les permissions. Comme tous les éléments du système Linux, les IPC possèdent un propriétaire et des droits d'accès. Chaque IPC possède une structure ***ipc_perm*** qui contient les données relatives aux permissions. On y trouve les champs standards identifiant le propriétaire, le créateur de la ressource, et les droits d'accès. Ces droits sont importants puisqu'ils permettent aux autres processus de pouvoir lire des données ou d'en écrire,

- ✓ **octets** et **nattch** correspondent à la taille de la mémoire partagée et au nombre d'attachements pour cette mémoire partagée,
- ✓ **nsems** correspond au nombre de sémaphore de l'ensemble,
- ✓ **octets utilisés** et **messages** correspondent à la place utilisée et au nombre de messages contenus pour une queue de message.

II/Les mémoires partagées

Les processus ont souvent besoin de partager des données. Le system Unix fournit à cet effet un ensemble de routines permettant de créer et de gérer les segments de mémoires partagées.

Un segment de mémoire partagée est identifié de manière externe par un nom qui permet à tout processus possédant les droits d'accéder à ce segment. Lors de la création ou de l'accès à un segment de mémoire partagée, un numéro interne est fourni par le système.

Parmi les mécanismes de communication entre processus, l'utilisation de mémoire partagée est la technique la plus rapide car il n'y a pas de copie des données transmises. Ce procédé est donc parfaitement adapté au partage de gros volumes de données entre processus distincts.

Les segments de mémoire partagée ont une existence indépendante des processus. Un processus pourra demander le rattachement d'un segment de mémoire partagée à son espace d'adressage et donc accéder aux données de ce segment.

II-1/ Création de mémoire partagée

La fonction ***int shmget(key t clé, int size, int shmflg)*** permet de créer un segment de mémoire partagée.

- ➢ Le paramètre ***clé*** est une clé qui peut évidemment être obtenue grâce à la fonction ***ftok***.
- ➢ Le paramètre ***size*** correspond à la taille en octets du segment.
- ➢ Le paramètre ***shmflg*** correspond à une combinaison de l'option IPC_CREAT (pour la création) et de droits d'accès à ce segment (propriétaire, groupe et autres).

La valeur retournée correspond à l'identificateur de ce segment et sera utilisée par les fonctions de gestion.

II-2/ attachement/détachement à un segment de mémoire partagée

Lorsqu'un segment de mémoire partagé est créé, il n'est pas encore accessible par un processus. Il va falloir que le processus rattache ce segment dans son espace d'adressage.

La fonction ***char *shmat (int shmid, char *shmaddr, int shmflg)*** permet à un processus de s'attacher au segment et donc d'avoir accès aux données.

- Le premier paramètre correspond à l'identificateur du segment (retourné par la fonction *shmget*),
- le second paramètre est l'adresse de la mémoire partagée. Si on passe NULL comme paramètre, c'est le système qui choisit l'adresse d'implantation.
- Le dernier paramètre correspond aux options d'attachement : si on précise une adresse (2nd paramètre), il faut ajouter l'option SHM_RND.

Il est possible de restreindre les droits d'accès en lecture seule à un segment de mémoire partagée en précisant SHM_RDONLY même si les droits donnés lors de la création autorisent l'écriture.
Une tentative d'écriture dans un segment de mémoire partagée avec l'option SHM_RDONLY entraînera la génération d'un signal SEGV.
Il n'existe pas d'attachement en écriture seule.
En cas de succès, la valeur retournée par cette fonction est un pointeur qui permet l'accès aux données.

La fonction ***int shmdt (char *shmaddr)*** permet de se détacher d'un segment de mémoire partagée. Le paramètre attendu est celui retourné par la fonction ***shmat***. En cas de succès, cette fonction renvoie 0.

Remarques :

- ✓ Un segment est automatiquement détaché lorsqu'un processus se termine,
- ✓ Le détachement d'un segment de mémoire partagée ne supprime pas ce dernier.

II-3/ Contrôle d'un segment de mémoire partagée

La fonction ***int shmctl(int shmid, int cmd, struct shmid_ds *buf)*** permet en fonction du paramètre ***cmd*** de contrôler un segment de mémoire partagée. Le premier paramètre correspond à l'identificateur du segment (retourné par la fonction ***shmget***). Les différentes commandes possibles sont :

- IPC_RMID : qui permet de détruire un segment de mémoire partagée. Ce segment ne sera effectivement détruit que lorsque le nombre d'attachements sera nul,
- IPC_STAT : qui permet de récupérer les informations d'un segment de mémoire partagée. Ces informations sont mémorisées dans la structure shmid_ds pointée par buf,
- IPC_SET : qui permet d'appliquer les paramètres de la structure shmid_ds pointée par buf sur le segment de mémoire partagée (sous réserve d'avoir les droits de le faire !).

Le troisième paramètre est un pointeur sur une structure *shmid*_ds qui permet d'appliquer ou de récupérer les paramètres. Cette structure a la composition suivante :

```
struct shmid_ds {
        struct ipc_perm shm_perm;        /* operation perms */
        int     shm_segsz;               /* size of segment (bytes) */
        time_t  shm_atime;               /* last attach time */
        time_t  shm_dtime;               /* last detach time */
        time_t  shm_ctime;               /* last change time */
        unsigned short  shm_cpid;        /* pid of creator */
        unsigned short  shm_lpid;        /* pid of last operator */
        short   shm_nattch;              /* no. of current attaches */

                                         /* the following are private */

        unsigned short   shm_npages;     /* size of segment (pages) */
        unsigned long   *shm_pages;      /* array of ptrs to frames -> SHMMAX */
        struct vm_area_struct *attaches; /* descriptors for attaches */
};
```

La structure *ipc*_perm a la composition suivante :

```
struct ipc_perm {
     uid_t           uid;    /* owner's user id */
     gid_t           gid;    /* owner's group id */
     uid_t           cuid;   /* creator's user id */
     gid_t           cgid;   /* creator's group id */
     unsigned short  mode;   /* access modes */
     unsigned short  seq;    /* slot usage sequence number */
     key_t           key;    /* key */
};
```

II-4/Exemples

Exemple1 : CreerZDC.c

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

int shmCreate(char *nom_cle, int taille);
char *shmAttach(int semId);
int shmDetach(char *pZDC);

int main(int argc, char *argv[])
{
```

```
int shmId;
char *pZDC, *pCopie;
int i;

        if ((shmId = shmCreate(argv[0],52)) == -1)
                return EXIT_FAILURE;
        printf("ZDC créée shmId = %d\n",shmId);
        if ((pZDC = shmAttach(shmId)) == NULL)
        {
                return EXIT_FAILURE;
        }
        pCopie = pZDC;
        printf("ZDC attachée Ptr = 0x%X\n",(int)pZDC);
        for (i=0;i<26;i++)
        {
                *pZDC++ = 'A' + i;
        }
        for (i=0;i<26;i++)
        {
                *pZDC++ = 'a' + i;
        }
        printf("Avant détachement\n");
        system("ipcs");
        if (shmDetach(pCopie) == -1)
        {
                perror("shmdt");
                return EXIT_FAILURE;
        }
        printf("Après détachement\n");
        system("ipcs");
        return EXIT_SUCCESS;
}

int shmDetach(char *pZDC)
{
        return shmdt(pZDC);
}

char *shmAttach(int semId)
{
char *Ptr;

        if ((Ptr = shmat(semId,0,0)) == (char *)-1)
        {
                perror("shmat");
                return NULL;
        }
        return Ptr;
}

int shmCreate(char *nom_cle,int taille)
{
key_t cle;
int shmId;
```

```
        if ((cle = ftok(nom_cle,0)) == -1)
        {
                perror("ftok");
                return -1;
        }
        printf("clé : 0x%X\t",cle);
        if ((shmId = shmget(cle,taille,IPC_CREAT | IPC_EXCL | 0600)) == -1)
        {
                printf("Elle existe !\n");
                if ((shmId = shmget(cle,taille,IPC_EXCL)) == -1)
                {
                        perror("shmget");
                        return -1;
                }
        }
        return shmId;
}
```

Exemple2 : LireZDC.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>         /* pour sleep */

char *shmAttach(int semId);
int shmDetach(char *pZDC);

int main(int argc, char *argv[])
{
char *pZDC;
char *pZone;
char *pCopie;
int i;

        if (argc != 2)
        {
                printf("Il faut donner l'identificateur de la mémoire partagée\n");
                return EXIT_FAILURE;
        }
        if (atoi(argv[1]) < 0)
        {
                printf("L'argument ne ressemble pas à un identificateur de mémoire partagée\n");
                return EXIT_FAILURE;
        }
        if ((pZDC = shmAttach(atoi(argv[1]))) == NULL)
        {
                printf("Impossible de s'attacher !\n");
                return EXIT_FAILURE;
```

```
	}
	pCopie = pZDC;
	pZone = (char *)malloc(52);
	if (pZone == NULL)
	{
		perror("malloc");
		shmDetach(pZDC);
		return EXIT_FAILURE;
	}
	for (i=0;i<52;i++)
	{
		pZone[i] = *pCopie++;
	}
	for (i=0;i<52;i++)
	{
		printf("%c",pZone[i]);
		if (i == 25)
			printf("\n");
	}
	printf("\n");
	printf("Ecriture d'une etoile\n");
	pZDC[0] = '*';
	pCopie = pZDC;
	for (i=0;i<52;i++)
	{
		pZone[i] = *pCopie++;
	}
	for (i=0;i<52;i++)
	{
		printf("%c",pZone[i]);
		if (i == 25)
			printf("\n");
	}
	printf("\n");
	if (shmDetach(pZDC) == -1)
	{
		perror("shmdt");
		free(pZone);
		return EXIT_FAILURE;
	}
	free(pZone);
	return EXIT_SUCCESS;
}

int shmDetach(char *pZDC)
{
	return(shmdt(pZDC));
}

char *shmAttach(int semId)
{
char *Ptr;

	if ((Ptr = shmat(semId,0,0)) == (char *)-1)
/*	if ((Ptr = shmat(semId,0,SHM_RDONLY)) == (char *)-1)*/
```

```
                {
                perror("shmat");
                return NULL;
        }
        return Ptr;
}
```

Exemple3 : DetruireZDC.c

#include <stdio.h>

```
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>          /* pour sleep */

int shmDestroy(int shmId);

int main(int argc, char *argv[])
{

        if (argc != 2)
        {
                printf("Il faut donner l'identificateur de la mémoire partagée\n");
                return EXIT_FAILURE;
        }
        if (atoi(argv[1]) < 0)
        {
                printf("L'argument ne ressemble pas à un identificateur de mémoire partagée\n");
                return EXIT_FAILURE;
        }
        return shmDestroy(atoi(argv[1]));
}

int shmDestroy(int shmId)
{
        if (shmctl(shmId,IPC_RMID,NULL) == -1)
        {
                perror("shmctl");
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
```

III/Les sémaphores

III-1/Introduction :

Si plusieurs processus veulent accéder à une même ressource, des problèmes risquent d'apparaître. Imaginez plusieurs processus rédacteurs sur une mémoire partagée ou un processus rédacteur qui intervient pendant un processus lecteur !

Pour éviter ces problèmes, on utilise des sémaphores qui permettent entre autres l'exclusivité d'une ressource partagée à un seul processus.

Ceci fait apparaître des **sections critiques** au début desquelles on prend le sémaphore et à la fin desquelles on vend le sémaphore. Dans le cas de l'exclusivité, si un autre processus a déjà pris le sémaphore, le processus qui veut le prendre sera mis en attente.

Un sémaphore permet de protéger une variable (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées dans un environnement de programmation concurrente.

Le sémaphore a été inventé par Edsger Dijkstra.

Définitions :

- ***Section Critique :*** C'est une partie de code telle que 2 processus ne peuvent s'y trouver au même instant.
- ***Exclusion mutuelle :*** Une ressource est en exclusion mutuelle si seul un processus peut utiliser la ressource à un instant donné.
- ***Conditions de fonctionnement :*** Plusieurs conditions sont nécessaires pour le bon fonctionnement de processus coopérants :
  - ✓ Deux processus ne peuvent être, en même temps, en section critique,
  - ✓ Aucune hypothèse n'est faite, ni sur la vitesse relative des processus, ni sur le nombre de processeurs,
  - ✓ Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus,
  - ✓ Aucun processus ne doit attendre "trop longtemps" avant de pouvoir entrer en section critique

Le masquage des interruptions :

- L'idée consiste à empêcher la commutation de processus pendant l'exécution de la section critique. Ce résultat est atteint en masquant les interruptions qui provoquent le changement d'allocation du processeur. Le traitement des interruptions est alors différé :
  - Masquer_Interruption;
  - SectionCritique;
  - Restaurer_Interruption;
- Le masquage des interruptions est une technique utile dans le noyau, mais inappropriée pour les processus utilisateurs

Un sémaphore S *est une variable entière qui n'est accessible qu'au travers de trois opérations Init, P et V.*

● La valeur d'un sémaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres. S'il n'y a qu'une ressource, un sémaphore binaire avec les valeurs 0 ou 1 est utilisé.

● Les opérations doivent être indivisibles (atomiques), ce qui signifie qu'elles ne peuvent pas être exécutées plusieurs fois de manière concurrente. Un processus qui désire exécuter une opération qui est déjà en cours d'exécution par un autre processus doit attendre que le premier termine.

Les opérations :

- L'opération **Init (S, valeur)** est seulement utilisée pour initialiser le sémaphore S avec une valeur. Cette opération ne doit être réalisée qu'une seule et unique fois.
- L'opération **P(S)** (du néerlandais *Proberen* signifiant tester et en français "Puis-je ?" ou éventuellement Prise ou Prendre) est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant.

```
SI (S > 0)
        ALORS S --;
SINON (attendre sur S)
FSI
```

- L'opération **V(S)** (du néerlandais *Verhogen* signifient tester, incrémenter et en français "Vas- y !" ou éventuellement Vente ou Vendre) est l'opération inverse. Elle rend simplement une ressource disponible à nouveau après que le processus ait terminé de l'utiliser

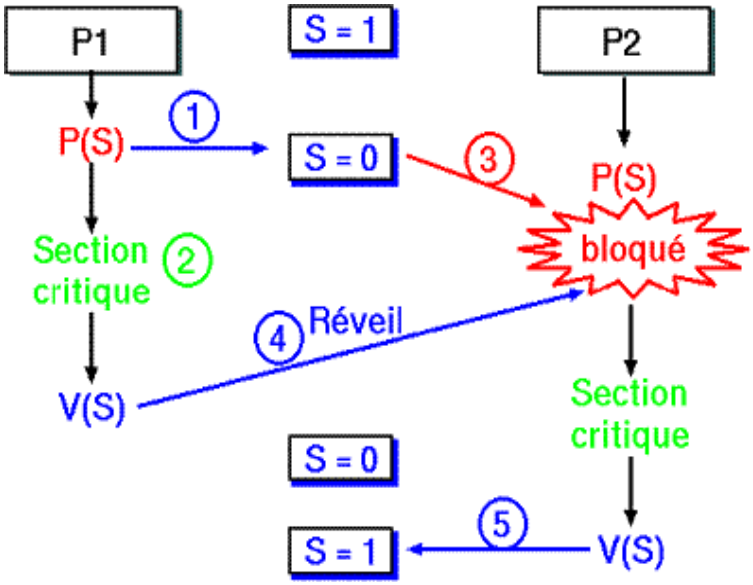
```
S++;
SI (des processus sont en attente sur S)
        ALORS laisser l'un deux continuer (reveil)
FSI
```

Sémaphore binaire :

Un sémaphore binaire est un sémaphore qui est initialisé avec la valeur 1. Ceci a pour effet de contrôler l'accès à une ressource unique. Le sémaphore binaire permet l'exclusion mutuelle (mutex) : une ressource est en exclusion mutuelle si un seul processus peut utiliser la ressource à un instant donné

Accès à une section critique :

L'accès à la section critique en utilisant un sémaphore binaire peut être représenté par :



III-2/Création de sémaphore :

La fonction ***int semget (key_t key, int nsems, int semflg)*** permet de créer un ensemble de sémaphores ou d'utiliser un ensemble de sémaphores déjà existant.

- Le paramètre ***key*** est une clé qui peut évidemment être obtenue grâce à la fonction *ftok*.
- Le paramètre ***nsems*** représente le nombre de sémaphores désiré (d'où le nom d'ensemble de sémaphores).
- Le dernier paramètre ***semflg*** correspond à des options :

  - IPC_CREAT : qui crée l'ensemble de sémaphore s'il n'existe pas,
  - IPC_EXCL qui en conjonction avec IPC_CREAT déclenchera une erreur si l'ensemble de sémaphores existe déjà (et donc précisée seule permet de retrouver l'identificateur de cet ensemble),
  - Des droits d'accès (propriétaire, groupe et autres).

La valeur retournée correspond à l'identificateur de l'ensemble de sémaphores.

Exemple : semget

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

int main(int argc, char * argv[])
{
key_t cle;
int semId;

    if (argc != 2)
    {
        printf("Il faut passer la valeur initiale en parametre !\n");
        return EXIT_FAILURE;
    }
    if (atoi(argv[1]) <= 0)
    {
        printf("Il faut une valeur strictement positive\n");
        return EXIT_FAILURE;
    }
    if ((cle = ftok("/etc",0)) == -1)
    {
        perror("ftok");
        return EXIT_FAILURE;
    }
    printf("cle : 0x%X\t",cle);
    if ((semId = semget(cle,1,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        if ((semId = semget(cle,1,IPC_EXCL)) == -1)
        {
            perror("semget");
            return EXIT_FAILURE;
        }
        printf("Le semaphore existait deja j'ai obtenu semId = %d\n",semId);
    }
    else
        printf("semId = %d\n",semId);
    if (semctl(semId,0,SETVAL,atoi(argv[1])) == -1)
    {
        perror("semctl SETVAL");
        semctl(semId, 0, IPC_RMID, NULL);
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

III-3/Prendre/vendre un sémaphore :

La fonction ***int semop (int semid, struct sembuf *sops, unsigned nsops)*** permet de prendre ou de vendre tout ou partie d'un ensemble de sémaphores identifié par ***semid***. Ces opérations sont réalisées **atomiquement**.

Pour prendre ou vendre un sémaphore, il faut remplir une structure ***sembuf*** (pointée par ***sops***). Cette structure est composée des champs suivants :

- short ***sem_num*** : qui correspond au numéro du sémaphore (0 = premier),
- short ***sem_op*** : qui correspond à l'opération à réaliser : – 1 pour prendre et +1 pour vendre (valeurs les plus courantes),
- short ***sem_flg*** : qui correspond aux options : ***IPC_NOWAIT*** pour réaliser une opération non bloquante ou ***SEM_UNDO*** qui annulera l'action lorsque le processus se terminera.

Le dernier paramètre correspond au nombre d'opérations à effectuer. En cas de succès, cette fonction retourne 0.

Exemple : Prendre/vendre

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

int main(int argc, char * argv[])
{
struct sembuf Param;

    if (argc != 2)
    {
        printf("Il me faut un identificateur de semaphore !\n");
        return EXIT_FAILURE;
    }
    if (atoi(argv[1]) <= 0)
    {
        printf("Identificateur non valide !\n");
        return EXIT_FAILURE;
    }
    Param.sem_num = 0;
    Param.sem_op = -1;

    if (semop(atoi(argv[1]),&Param,1) == -1)
    {
        perror("semop");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

III-4/Contrôle de sémaphore :

La fonction ***int semctl (int semid, int semno, int cmd, union semun arg)*** permet de consulter ou modifier un ensemble de sémaphores. Cet ensemble est précisé par le paramètre ***semid***. Le paramètre ***semno*** permet de préciser un sémaphore de l'ensemble. Le paramètre ***arg*** permet d'avoir les informations ou de donner les modifications à réaliser en fonction du paramètre ***cmd***. Les différentes commandes possibles sont les suivantes :

- ***GETALL*** : renvoie la valeur ***semval*** de chaque sémaphore de l'ensemble dans le tableau ***arg.array***,
- ***GETNCNT*** : renvoie le nombre de processus en attente d'une incrémentation du champ ***semval*** du sémaphore ***semno***,
- ***GETPID*** : renvoie le PID du processus ayant exécuté le dernier appel système ***semop*** pour le sémaphore ***semno***,
- ***GETVAL*** : renvoie la valeur du champ ***semval*** du sémaphore ***semno***,
- ***GETZCNT*** : renvoie le nombre de processus attendant que le champ ***semval*** du sémaphore ***semno*** soit nul,
- ***SETALL*** : positionne le champ ***semval*** de tous les sémaphores de l'ensemble en utilisant le tableau ***arg.array***. Les processus en attente sont réveillés si ***semval*** devient 0 ou augmente,

- ***SETVAL*** : place la valeur ***arg.val*** dans le champ ***semval*** du sémaphore ***semno***. Les processus en attente sont réveillés si ***semval*** devient 0 ou augmente,
- ***IPC_STAT*** : copie dans la structure pointée par ***arg.buf*** la structure de données concernant l'ensemble de sémaphores,
- ***IPC_SET*** : fixe la valeur de certains champs de la structure ***semid_ds*** pointée par ***arg.buf*** dans la structure de contrôle de l'ensemble de sémaphores,
- ***IPC_RMID*** : supprime immédiatement l'ensemble de sémaphores et réveille tous les processus en attente.

Un ensemble de sémaphores est décrit par la structure suivante :

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

struct semid_ds{
        struct ipc_perm sem_perm; /* Permissions d'accès */
        struct sem *sem_base;
        u_short sem_nsems; /* Nombre de sémaphores dans l'ensemble */
        time_t sem_otime;  /* Heure dernier semop() */
        time_t sem_ctime;  /* Heure dernier changement */
        };
```

Dans tous les cas, le processus appelant doit avoir des privilèges d'écriture sur le jeu de sémaphores. La valeur retournée dépend du paramètre *cmd*.

Exemple : semctl

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

int main(int argc, char * argv[])
{
struct semid_ds info;

    if (argc != 2)
    {
        printf("Il me faut un identificateur de semaphore !\n");
        return EXIT_FAILURE;
    }
    if (atoi(argv[1]) <= 0)
    {
        printf("Identificateur non valide !\n");
        return EXIT_FAILURE;
    }
    if (semctl(atoi(argv[1]), 29, IPC_STAT, &info) == -1)
    {
        perror("semctl");
        return EXIT_FAILURE;
    }
    printf("Clé : 0x%x\n",info.sem_perm.__key);
    printf("UID propriétaire : %d GID propriétaire : %d\n",info.sem_perm.uid, info.sem_perm.gid);
    printf("droits : %o\n",info.sem_perm.mode & 0777);
    printf("Nombre de sémaphore dans l'ensemble : %ld\n", info.sem_nsems);
    printf("Valeur du semaphore : %d\n",semctl(atoi(argv[1]),0,GETVAL,NULL));
    printf("Nombre de processus etant bloque en attente de ce semaphore : %d\n",semctl(atoi(argv[1]),0,GETNCNT,NULL));
    return EXIT_SUCCESS;
}
```

III-5/Rendez-vous de processus : opération Z :

Le champ ***sem_op*** de la structure ***sembuf*** qui est passée à la fonction ***semop*** permet de prendre ou de vendre un sémaphore (habituellement valeur -1 ou +1). Si on met ce champ à la valeur 0, le processus attendra que la valeur du sémaphore (***semval***) soit égale à 0. Avec cette possibilité, il est donc possible de créer un rendez-vous de processus. Pour cela, on crée un sémaphore initialisé à une valeur correspondant au nombre de processus devant être synchronisés : N. Chaque processus utilisera la fonction ***semop*** pour prendre le sémaphore puis la fonction ***semop*** avec la valeur ***sem_op*** à ***0*** et lorsque les N processus auront réalisé cette opération, ils seront tous libérés et auront donc été synchronisés sur le plus lent.

Exemple : Rendez-vous

```c
int semZ(int semId)
{
struct sembuf Param;

    Param.sem_num = 0;
    Param.sem_op = -1;
    Param.sem_flg = SEM_UNDO;
    if (semop(semId,&Param,1) == -1)
    {
        perror("semop");
        return EXIT_FAILURE;
    }
    Param.sem_op = 0;
    if (semop(semId,&Param,1) == -1)
    {
        perror("semop");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```c
int semCreate(char *nom_cle, int val)
{
key_t cle;
int semId;

    if (val < 0)
    {
        printf("Il faut passer une valeur initiale >= 0 en parametre !\n");
        return EXIT_FAILURE;
    }
    if ((cle = ftok(nom_cle,0)) == -1)
    {
        perror("ftok");
        return EXIT_FAILURE;
    }
    if ((semId = semget(cle,1,IPC_CREAT | IPC_EXCL | 0600)) == -1)
    {
        if ((semId = semget(cle,1,IPC_EXCL)) == -1)
        {
            perror("semget");
            return EXIT_FAILURE;
        }
        printf("Le semaphore existait deja j'ai obtenu semId = %d\n",semId);
    }
    else
    {
        printf("semId = %d\n",semId);
        if (semctl(semId,0,SETVAL,val) == -1)
        {
            perror("semctl");
            return EXIT_FAILURE;
        }
    }
    return semId;
}
```

```c
int semPrendre(int semId)
{
struct sembuf Param;

    Param.sem_num = 0;
    Param.sem_op = -1;
    Param.sem_flg = SEM_UNDO;
    if (semop(semId,&Param,1) == -1)
    {
        perror("semop");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

int semVendre(int semId)
{
struct sembuf Param = {0, 1, SEM_UNDO};

    if (semop(semId,&Param,1) == -1)
    {
        perror("semop");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

```c
int main(int argc, char *argv[])
{
int semId;
int i;
pid_t pid[5];

    if (argc != 2)
    {
        printf("Il me faut le nombre de processus a creer\n");
        return EXIT_FAILURE;
    }
    if ((atoi(argv[1]) < 2) | (atoi(argv[1]) > 5))
    {
        printf("La valeur doit etre > 2 et < 5\n");
        return EXIT_FAILURE;
    }
    semId = semCreate(argv[0],atoi(argv[1]));
    printf("GETZCNT =  %d\n",semctl(semId,0,GETZCNT,NULL));
    printf("GETVAL = %d\n\n",semctl(semId,0,GETVAL,NULL));
    for(i=0;i<atoi(argv[1]);i++)
    {
        pid[i] = fork();
        switch(pid[i])
        {
            case -1 :
            {
                perror("fork");
                semctl(semId,0,IPC_RMID,NULL);
                return EXIT_FAILURE;
            }
            case 0 :
            {
                semV(semId);
                printf("Je suis le fils %d de PID : %d je suis debloque\n",i+1,getpid());
                sleep(1);
                return EXIT_SUCCESS;
            }
            default :
            {
                sleep(1);
                printf("GETZCNT =  %d\n",semctl(semId,0,GETZCNT,NULL));
                printf("GETVAL = %d\n\n",semctl(semId,0,GETVAL,NULL));
                sleep(1);
            }
        }
    }
    printf("attente de terminaison des fils\n");
    for(i=0;i<atoi(argv[1]);i++)
    {
        printf("Fils de PID %d termine\n",wait(NULL));
    }
    printf("GETZCNT =  %d\n",semctl(semId,0,GETZCNT,NULL));
    printf("GETVAL = %d\n",semctl(semId,0,GETVAL,NULL));
    printf("Destruction du semaphore\n");
    semctl(semId,0,IPC_RMID,NULL);
    return EXIT_SUCCESS;
}
```

IV/ les queues (ou files) de messages :

IV-1/Introduction

Une file de messages est une liste chaînée gérée par le noyau dans laquelle un processus peut déposer des données (messages) ou en extraire. Elle correspond au concept de boîte aux lettres.

Un message est une structure comportant un nombre entier (le type du message) et une suite d'octets de longueur arbitraire, représentant les données proprement dites.

Les messages étant typés, il est possible de les lire dans un ordre différent de celui de leur insertion. Le 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 au processus 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.

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 ainsi que celle de la file.

La manipulation des files de messages suppose la définition d'une structure spécifique à l'application développée sur le modèle donné par msgbuf.

```
struct msgbuf
{
        long int mtype; //type du message, doit être >0
        char mtext[1];  //texte du message
};
```

Le premier champ de type long int (mais > 0) détermine le type de message et permet de partitionner les messages. Une queue de messages pourra lire les messages d'un type particulier grâce à ce paramètre (d'où ce nom de partitionnement). Les autres membres de cette structure sont laissés au choix du programmeur à la seule condition de ne pas réaliser d'indirection (c'est-à-dire utiliser des pointeurs).

IV-2/Création d'une file de messages

La fonction ***int msgget (key_t key, int msgflg)*** permet de créer une file de messages ou d'utiliser une file de messages existante.

Le paramètre *key* est une clé qui peut évidemment être obtenue grâce à la fonction ***ftok.*** Le paramètre ***msgflg*** correspond à des options :

- **IPC_CREAT** : qui crée la file de messages si elle n'existe pas,

- **IPC_EXCL** qui en conjonction avec IPC_CREAT déclenchera une erreur si la file de messages existe déjà (et donc précisée seule permet de retrouver l'identificateur de cette file de messages),
- Des droits d'accès (propriétaire, groupe et autres) peuvent aussi être spécifiés en options.

La fonction ***msgget*** retourne l'identificateur de la file de message.

IV-3/Lecture d'un message dans une file de message

La fonction

ssize msgrcv (int msqid, struct msgbuf * msgp, size_t msgsz, long msgtyp, int msgflg)

msgrcv permet de lire un message dans la file de messages d'identificateur ***msqid.***

Le message lu, de taille ***msgsz*** (le long int définissant le type est non compris), sera mémorisé dans la structure pointée par ***msgp***.

Le type du message sera spécifié par le dernier paramètre : ***msgtyp***.

Si ***msgtyp*** vaut 0, le premier message est lu. Si ***msgtyp*** est supérieur à 0, alors le premier message de type ***msgtyp*** est extrait de la file. Si ***msgtyp*** est inférieur à 0, le premier message de la file avec un type inférieur ou égal à la valeur absolue de ***msgtyp*** est extrait. En cas de succès, cette fonction renvoie le nombre d'octets lus.

IV-4/ Ecriture d'un message dans une file de message

La fonction

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

msgsnd permet d'écrire le message, de taille ***msgsz***, pointé par ***msgp*** dans la file d'identificateur ***msqid***. En cas de succès, cette fonction renvoie 0.

*Pour les deux fonctions, **msgrcv** et **msgsnd**, il est possible de choisir des options avec le paramètre **msgflg** (IPC_NOWAIT, MSG_NOERROR par exemple).*

IV-5/Contrôle d'une file de messages

La fonction :

int msgctl (int msqid, int cmd, struct msqid ds *buf);

msgctl permet de consulter ou modifier les caractéristiques de la file de messages d'identificateur ***msqid***. Le paramètre ***cmd*** précise la commande réalisée parmi celles-ci :

- ***IPC_STAT*** : qui copie les informations de la file de messages dans la structure pointée par ***buf***,
- ***IPC_SET*** : qui fixe les informations de la structure pointée par ***buf*** dans la file de messages,
- ***IPC_RMID*** : qui supprime immédiatement (à condition d'avoir les droits) la file de messages.

En cas de succès, cette fonction retourne 0.

La description d'une file de message se fait grâce à la structure *msqid_ds* définit par :

```
struct msqid_ds
{
    struct ipc_perm msg_perm;     /* Propriétaire et permissions */
    time_t  msg_stime;    /* Heure du dernier msgsnd(2) */
    time_t  msg_rtime;    /* Heure du dernier msgrcv(2) */
    time_t  msg_ctime;    /* Heure de dernière modification */
    unsigned long   __msg_cbytes; /* Nombre actuel d'octets dans la file(non standard) */
    msgqnum_t   msg_qnum;     /* Nombre actuel de messages dans la file */
    msglen_t    msg_qbytes;   /* Nombre maximal d'octets autorisés dans la file */
    pid_t   msg_lspid;    /* PID du dernier msgsnd(2) */
    pid_t   msg_lrpid;    /* PID du dernier msgrcv(2) */
};
```

IV-6/Exemples

Exemple1 : Création d'une file de messages

```c
#define TAILLE 20
int main (int argc, char *argv[])
{
    typedef struct{
        long mtype;
        char Message[TAILLE];
        }MSGRQ;

    key_t cle;
    int msgId;
    MSGRQ requete;
    cle = ftok("/etc/passwd",0);
    if (cle == -1){
        perror("ftok");
        return EXIT_FAILURE;
    }
    msgId = msgget(cle,IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    /* existent aussi : S_IRGRP, S_IWGRP, S_IROTH et S_IWOTH */
    if (msgId == -1){
        msgId = msgget(cle,IPC_EXCL);
        if (msgId == -1)        {
            perror("msgget");
            return EXIT_FAILURE;
        }
        printf("La MSG existait deja identificateur : %d\n",msgId);
    }
    if (argc != 1)  requete.mtype = atoi(argv[1]);
    else    requete.mtype = 1;
    if (argc > 2)   strcpy(requete.Message, argv[2]);
    else    strcpy(requete.Message,"Message generique");
    if ((msgsnd(msgId,&requete,sizeof(MSGRQ)-sizeof(long),0)) == -1){
        perror("msgsnd");
        return EXIT_FAILURE;
    }
    printf("MSG creee avec la cle %x\n",cle);
    printf("Identificateur de la MSG : %d\n",msgId);
    return EXIT_SUCCESS;
}
```

Exemple2 : lecture d'un message dans la file de messages

```
#define TAILLE 20
typedef struct
{
    long mtype;
    char Message[TAILLE];
}MSGRQ;
int main (int argc, char *argv[]){
    MSGRQ ReqRecu;
    key_t cle;
    int msgId;
    if (argc != 2)  {
        printf("Il faut 1 parametres : le type de message a lire\n");
        return EXIT_FAILURE;
    }
    cle = ftok("/etc/passwd",0);
    if (cle == -1)  {
        perror("ftok");
        return EXIT_FAILURE;
    }
    msgId = msgget(cle,IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    if (msgId == -1)    {
        msgId = msgget(cle,IPC_EXCL);
        if (msgId == -1)    {
            perror("msgget");
            return EXIT_FAILURE;
        }
        printf("La MSG existait deja identificateur : %d\n",msgId);
    }
    if ((msgrcv(msgId,&ReqRecu,sizeof(MSGRQ)-sizeof(long),atoi(argv[1]),0)) == -1)
    {
        perror("msgrcv");
        return EXIT_FAILURE;
    }
    printf("Lecture des messages de type %d sur la MSG d'identificateur %d :\n\n",atoi(argv[1]),msgId);
    printf("type : %ld\n",ReqRecu.mtype);
    printf("Message : %s\n",ReqRecu.Message);
    return EXIT_SUCCESS;
}
```

Exemple3 : Informations sur la file de message

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

int main (int argc, char *argv[])
{

key_t cle;
int msgId;
struct msqid_ds info;

    cle = ftok("/etc/passwd",0);
    if (cle == -1)  {
        perror("ftok");
        return EXIT_FAILURE;
    }
    msgId = msgget(cle,IPC_EXCL);
    if (msgId == -1)    {
        printf("La queue n'existe pas\n");
        return EXIT_FAILURE;
    }
    if (msgctl(msgId,IPC_STAT,&info) == -1) {
        perror("msgctl");
        return EXIT_FAILURE;
    }
    printf("uid = %d gid = %d droits = %o\n",info.msg_perm.uid, info.msg_perm.gid, info.msg_perm.mode);
    printf("Nombre de messages restants : %ld\n",info.msg_qnum);
    printf("Taille de tous les messages : %ld\n",info.__msg_cbytes);
    printf("Taille restante : %ld\n",info.msg_qbytes - info.msg_cbytes);
    return EXIT_SUCCESS;
}
```