3.Les ressources IPC : sémaphore, mémoire partagée et files de messages

3.1.Objectifs

La séance a pour but de familiariser l'étudiant avec les sémaphores, la mémoire partagée et les files de messages. La manipulation des sémaphores nous servira à synchroniser des processus. La mémoire partagée, permettra le partage de données entre différents processus. Les files de messages sera un moyen plus pratique pour échanger des messages entre différents processus.

3.2.Généralités sur les sémaphores

Les sémaphores sont des objets de communication inter processus (IPC) utilisés pour synchroniser des processus. Une variable entière '`s`' est associée à chaque sémaphore et est une image du nombre de processus pouvant obtenir un accès à la ressource. Une file d'attente est également liée au sémaphore et reprend la liste des processus souhaitant un accès à la ressource.

Deux opérations principales peuvent être effectuées sur un sémaphore :

- `down` (aussi appelé `proberen`) pour réserver, si possible, un accès à la ressource ;
- `up` (aussi appelé `verhogen`) pour rendre un accès à la ressource.

Le déroulement de ces opérations va être le suivant :

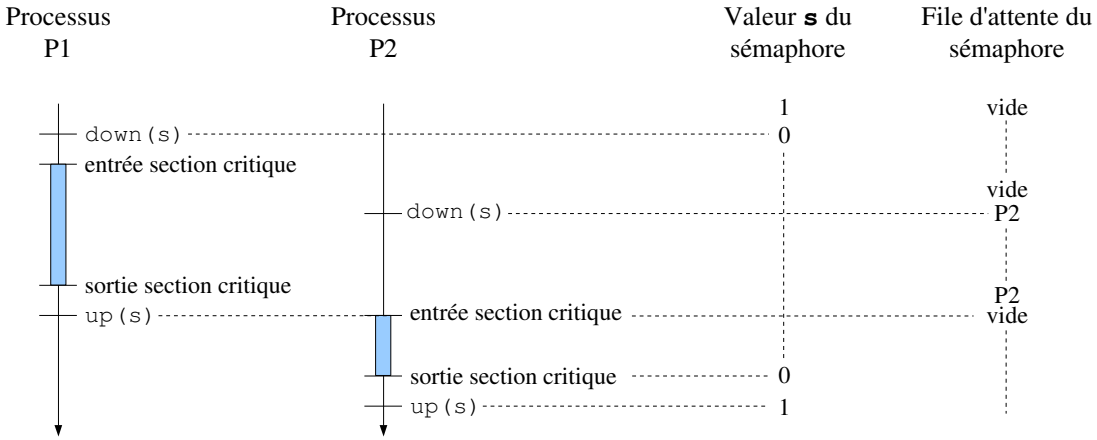
down		
si `s` strictement positif	→	`s--`
sinon	→	blocage du processus; mise du processus dans la file d'attente.

Déroulement de `down` dépend de `s`

up		
si file d'attente vide	→	`s++`
sinon	→	déblocage du premier processus de la file d'attente.

Déroulement de `up` dépend de la file d'attente

Exemple d'accès concurrent à une section critique par 2 processus, géré par un sémaphore **`s`**:



3.3.Généralités sur la mémoire partagée

La mémoire partagée est une zone de mémoire accessible par des processus distincts. Un processus doit d'abord rattacher la mémoire dans son espace d'adresses avant de pouvoir y accéder. Cette zone de mémoire étant commune à tout le système, une modification apportée par un processus est perçue par tous les autres.

Remarque: attention à ne pas confondre une variable en mémoire partagée avec une variable globale! Une variable globale est accessible par toutes les fonctions d'un processus mais est propre à un seul processus.

3.4.Généralités sur les files de messages

Les files de messages permettent l'échange de messages entre processus. Le fonctionnement est différent de celui des tubes car il n'est plus question d'un flux mais de messages de taille fixe. Les files sont en outre plus souples que les tubes, notamment pour la transmission sélective de messages. Nous pouvons ainsi définir plus facilement quel processus doit recevoir un message.

Un message est une structure composée :

- d'un entier long représentant le type de message à envoyer;
- des données utiles à intégrer au message.

Le type du message est choisi pour l'ensemble des processus échangeant les messages. Il permettra de hiérarchiser par priorité ou de multiplexer des messages dans une même file de messages.

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

3.5.Utilisation des ressources IPC

Sémaphores	Mémoire partagée	Files de messages

Inclusions de librairies:
`#include <sys/sem.h>`
`#include <sys/types.h>`
`#include <sys/ipc.h>`
	Inclusions de librairies:
	`#include <sys/shm.h>`
	`#include <sys/types.h>`
	`#include <sys/ipc.h>`
		Inclusions de librairies:
		`#include <sys/msg.h>`
		`#include <sys/types.h>`
		`#include <sys/ipc.h>`

Création d'un groupe de sémaphores: `int semget(int cle, int nsems, int flg);`	Création d'une mémoire partagée: `int shmget(int cle, int taille, int flg);`	Création d'une file de messages: `int msgget(int cle, int flg);`

Ces appels renvoient l'identifiant de la ressource IPC créée ou -1 en cas d'erreur.

Les arguments de ces appels sont:

- `cle` est l'identificateur (entier) de la ressource IPC; les valeurs possibles pour `cle` sont :
  - `IPC_PRIVATE` : la ressource créé n'est alors pas lié à une clé d'accès, et seul le processus propriétaire (le créateur de la ressource) et ses descendants ont accès à la ressource;
  - une valeur entière spécifiant la clé de la ressource (e.g. `1234`); cette clé permettra aux autres processus d'accéder à la ressource IPC créée.

- `flg` est un drapeau spécifiant les droits d'accès sur la ressource IPC créée; les valeurs possibles pour `flg` sont :
  - `IPC_CREAT` : indique la création d'une nouvelle ressource IPC si la clé fournie n'est pas déjà utilisée; dans le cas où la clé existe, l'appel renvoie l'identifiant de la ressource IPC existante associée à la clé;
  - `IPC_EXCL` : provoque l'échec de l'appel si la clé fournie est déjà utilisée par une autre ressource IPC du même type;
  - un entier spécifiant les droits d'accès à la ressource IPC.

• `nsems` indique le nombre de sémaphores à créer dans ce groupe.	• `taille` indique la taille en octets de la mémoire partagée.	
Exemple d'utilisation: `semid=semget(IPC_PRIVATE, 2, 0660);` Cette ligne va créer un ensemble de deux sémaphores en utilisant une clé privée. Ces sémaphores ne seront accessibles qu'au processus et à ses descendants. L'identifiant de ce groupe de sémaphore sera retourné dans `semid`.	Exemple d'utilisation: `shmid=shmget(1234, sizeof(int), IPC_CREAT|0660);` Cette ligne va créer une zone de mémoire partagée de la taille de 1 entier. Si un autre processus a déjà créé une mémoire partagée avec la clé 1234, la fonction `shmget` ne fait que renvoyer l'identifiant de cette zone mémoire. L'identifiant de cette mémoire partagée sera retourné dans `shmid`.	Exemple d'utilisation: `msgid=msgget(IPC_PRIVATE,0660);` Cette ligne va créer une file de messages. Cette file ne sera accessible qu'au processus et à ses descendants. L'identifiant de cette file de message sera retourné dans `msgid`.

Contrôle sur les sémaphores:
`int semctl(int semid, int numsem, int cmd, union semun valeur);`
 	
`union semun { int val; struct semid_ds *buf; unsigned short *array; struct seminfo *__buf; };`
	Contrôle sur les mémoires partagées:
	`int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
		Contrôle sur les files de messages:
		`int msgctl(int msqid, int cmd, struct msqid_ds *buf);`

Le premier paramètre passé à ces appels est chaque fois l'identifiant de la ressource IPC (retourné par l'appel à `semget`, `shmget` ou `msgget`).

Le paramètre `cmd` correspond à la commande que l'on souhaite réaliser sur la ressource IPC. Les valeurs possibles pour `cmd` sont :

- `IPC_STAT` : remplit la structure pointée par `buf` avec les informations de la ressource IPC;
- `IPC_SET` : permet de configurer les autorisations d'accès (le GID et l'UID du propriétaire, les permissions et la taille de la file) sur la ressource IPC à partir des information contenue dans `buf`;
- `IPC_RMID` : permet de supprimer la ressource IPC;

- `SETVAL` : initialisation du sémaphore `numsem` à une valeur déterminée (passée dans `valeur`);
- `GETVAL` : provoque le renvoie par la fonction `semctl()` de la valeur du sémaphore.

Exemple d'utilisation:

```
union semun sem_union;
sem_union.val = 1;
semctl(semid, 3, SETVAL,
sem_union);
```

Cette ligne va fixer à 1 la valeur du sémaphore numéro 3 (= 4ème sémaphore) dans l'ensemble de sémaphore `semid`.

Exemple d'utilisation:

```
shmctl(shmid, IPC_RMID,
NULL);
```

Cette ligne va supprimer la zone de mémoire partagée.

Exemple d'utilisation:

```
msgctl(msgid, IPC_RMID,
NULL);
```

Cette ligne va supprimer la file de messages.

Opérations `down` et `up` sur les sémaphores:

```
int semop(int semid, struct sembuf
*semops, size_t numsemops);

struct sembuf {
  // numéro du sémaphore
  short sem_num;

  // opération à réaliser
  //(down → -1 et up → +1)
  short sem_op;

  // paramètres spéciaux
  short sem_flg;
}
```

- `semid` : est l'identifiant de l'ensemble de sémaphores (valeur renvoyée par `semget()`);
- `semops` : est un pointeur vers un tableau de structures de type `struct sembuf`;
- `numsemops` : est le nombre d'opérations définies dans le tableau pointé par `semops`.

Valeurs possibles pour `semflg` :

- `IPC_NOWAIT` : indique que l'opération est non bloquante. Si l'opération ne peut pas être accomplie immédiatement, `semop` retourne -1 et la

Utilisation de la mémoire partagée:

Pour attacher le processus à la mémoire partagée:

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

Pour détacher le processus de la mémoire partagée:

```
shmdt(void *);
```

- `shmid` est l'identificateur (entier) de la zone mémoire (valeur renvoyée par `shmget()`);
- `shmaddr` est l'adresse à laquelle la mémoire doit être attachée au processus courant (si cette valeur vaut `NULL`, le système trouve automatiquement une adresse libre) ;
- `shmflg` est un drapeau binaire (`SHM_R | SHM_W`).

Envoi et réception de messages:

```
int msgsnd(int msgid, struct
msgbuf *msg, size_t taille, int
attributs);

int msgrcv(int msgid, struct
msgbuf *msg, size_t taille, long
mtype, int attributs);
```

Un message est défini par la structure suivante:

```
struct msgbuf {
  long mtype;
  char titre[TAILLE_TITRE];
  char texte[TAILLE_TEXTE];
};
```

Les paramètres utilisés par ces fonctions sont les suivants:

- `msgid`: est l'identifiant d'une file existante (renvoyé par `msgget()`);
- `msg`: est un pointeur vers une structure contenant le type de message et son contenu;
- `taille`: est la longueur en octets du message (type non compris);
- `mtype`: est le type de message à recevoir;
  - `mtype = 0` : indique que l'on veut recevoir le prochain message arrivant dans la file;

variable `errno`[1] vaut `EAGAIN`.

- `SEM_UNDO` : indique que l'opération doit être annulée si le processus ayant réalisé cette opération vient à se terminer.

Exemple d'utilisation:

```
struct sembuf sem;
sem.sem_num = 2;
sem.sem_op = -1;
sem.sem_flg = 0;
semop(semid, &sem, 1);
```

Ce code réalise une opération de type `down` sur le 3ème sémaphore du groupe `semid`.

Exemple d'utilisation:

```
int *writers;
writers = (int *)
shmat(shmid, NULL, SHM_R|
SHM_W);     // Rattachement
au processus
...
shmdt(writers);
```

- `mtype > 0` : indique que l'on veut recevoir le prochain message du type `mtype` arrivant dans la file;
- `mtype < 0` : réclame le premier message disponible ayant le plus petit type inférieur ou égal à la valeur absolue de `mtype`.

- `attributs` : généralement fixé à 0 pour obtenir les options par défaut; peut être est une combinaison binaire de flags suivants :
  - `IPC_NOWAIT` : rend les fonctions non-bloquantes. Si la file est pleine (`msgsnd`) ou s'il n'y a pas de message du type désiré dans la file (`msgrcv`), l'opération retourne -1 et la variable `errno`[2] vaut respectivement `EAGAIN` ou `ENOMSG`.
  - `MSG_EXCEPT` : réclame un message de n'importe quel type sauf mtype.

1 Code de la dernière erreur, situé dans le header `errno.h`
2 Cf. pied de page n°1

3.6.Exercice 1

Écrire un programme qui crée 2 processus fils. Chaque fils affiche 3 fois le message « `je suis le fils <pid>` », chaque affichage étant séparé par 1 seconde d'attente.

```
je suis le fils 123
je suis le fils 124
je suis le fils 123
je suis le fils 124
je suis le fils 123
je suis le fils 124
```

Dans un deuxième temps, on souhaite empêcher l'affichage simultané des deux fils, en considérant l'accès au terminal comme une ressource critique. Lorsqu'un fils démarre son exécution, il se réserve l'accès à la ressource critique (le terminal) et réalise l'affichage de ses 3 messages après quoi il « libère » l'accès au terminal; il n'est alors pas interrompu par le deuxième fils.

```
je suis le fils 123
je suis le fils 123
je suis le fils 123
je suis le fils 124
je suis le fils 124
je suis le fils 124
```

3.7.Exercice 2

Écrire un programme qui crée deux fils puis lit des entiers non nuls et transmet les négatifs au fils gauche et les positifs au fils droit. Lorsqu'il reçoit 0, le père indique aux deux fils la fin de la saisie par l'envoi d'un 0 dans le tube; les fils affichent alors la somme des nombres reçus.

Deux sémaphores seront ici utilisés pour synchroniser les 2 processus fils. Chaque fils attend l'autorisation d'accéder au tube (`down`), cette autorisation étant donnée par le processus père (`up`).

Remarques :

- Utiliser un seul tube;
- Ne pas utiliser les signaux.

3.8.Exercice 3

Créer un système client/serveur dans lequel le client introduit un texte d'une longueur fixée dans une mémoire partagée. Le serveur va changer le texte de la mémoire partagée en majuscules après quoi le client en affichera le contenu. Le tout sera synchronisé par des sémaphores.

Remarque : les codes du client et du serveur doivent se trouver dans des fichiers différents.

3.9.Exercice 4

Dans l'exercice 2, remplacer le tube et les sémaphores par une file de messages.

3.10.Références bibliographiques

1. Informatique temps réel - *Pierre Manneback*, 2006-2007 - Faculté Polytechnique de Mons.
2. Informatique de base - Notes de Laboratoire 2005-2006 - *Mohammed Benjelloun* – Faculté Polytechnique de Mons.
3. Programmation Linux - *Neil Matthew, Richard Stones* – Eyrolles 2000.
4. Programmation système en C sous Linux – *Christophe Blaess* - Éditions Eyrolles, 2005.