PROGRAMMATION SYSTÈME: TP 2

Les Sémaphores

L'objectif de ce TP est de mettre en pratique l'utilisation des sémaphores pour le partage de ressources entre plusieus processus.

1. Une introduction

Bien que le système Unix propose des primitives C pour traiter les sémaphores (cf. partie II), on se propose dans cette introduction de définir les fonctions `P` et `V` en utilisant un tube (pipe). En effet, comme vous l'avez vu dans les TP précédents, la lecture dans un tube vide est bloquante pour un processus. On peut donc imaginer le scénario suivant : soit une ressource accessible par au plus n processus à la fois et soit P un processus désirant y accéder. Tous les processus en exécution ont en commun un même tube T. Avant d'accéder à la ressource, P essaie de lire un caractère dans le tube T :

. si le tube T est vide, cela signifie que la ressource n'est pas disponible et P est bloqué,
. sinon la ressource est disponible, P peut y accéder et retire un caractère du tube pour le signaler,
. tout processus qui libère la ressource écrit un caractère dans le tube,
. au début le tube est rempli avec n caractères.

Exercice 1. A partir de la définition : `typedef int Semaphore[2]`, écrivez les fonctions suivantes :

. `initSem(Semaphore S, int N)` qui initialise le tube `S` avec `N` caractères quelconques,
. `P(Semaphore S)` qui lit un caractère dans `S`,
. `V(Semaphore S)` qui écrit un caractère quelconque dans `S`.

Exercice 2. On désire réaliser un programme qui prend en paramètre un entier n et qui affiche tous les entiers de 0 à n. Pour cela le programme se duplique, le père se charge de l'affichage des entiers pairs et le fils de l'affichage des entiers impairs. Vous utiliserez deux sémaphores intitulés `pair` et `impair` permettant de synchroniser l'exécution des deux processus.

2. Les fonctions IPC (Inter Processus Communication)

Dès lors que plusieurs processus fonctionnent en concurrence, se pose le problème de l'accès à une ressource partagée (un fichier par exemple). Comme nous l'avons vu dans la première partie les sémaphores peuvent être utilisés pour synchroniser les processus. Mais dans le cas où les processus ont besoin d'accéder à plusieurs ressources pour effectuer leur traitement, il se peut qu'ils forment une chaîne circulaire d'attente (*cf.* problème des philosophes) et que cela provoque un interblocage (dead-lock).

Pour éviter ce genre de problèmes, Unix propose la notion de groupe de sémaphores qui permet de réaliser les opérations par bloc. Les sémaphores Unix font en fait partie d'un ensemble de fonctions de communication entre processus n'ayant pas forcément de lien de parenté mais s'exécutant sur la même machine les IPC : Inter processus communication. Cet ensemble comprend :

(1) Les segments de mémoire partagées

(2) Les files de messages (cet aspect ne sera pas abordé)

Comme il s'agit de ressources partagées, les sémaphores sont utilisés pour réaliser l'exclusion mutuelle. Ces fonctionnalités étant proposées par le système deux fonctions sont disponibles dans le shell `ipcs` et `ipcrm`, pour vérifier l'état des ressources et pour les libérer.

2.1. **Préliminaires.** Les fonctions que nous allons manipuler possèdent de nombreuses options booléennes (drapeaux ou flag), certaines sont facultatives. Il n'est pas envisageable que chaque fonction prenne en paramètre toutes les options possibles (pourquoi ?). L'exercice suivant illustre une solution qui utilise les masques binaires.

Exercice 3. Le programme suivant (`exo3-testflag-squel.c`) permet de représenter et d'afficher une combinaison d'options de lecture, écriture et exécution sous la forme d'un seul paramètre entier de la fonction `afficheDroit()`. Chaque option est représentée par une constante. Compléter les parties manquantes.

Date: Novembre 2006.

```
#define LECTURE   /* A COMPLETER */
#define ECRITURE  /* A COMPLETER */
#define EXECUTION /* A COMPLETER */
typedef int droits;
void afficheDroit(droits D)
 {
  if (/* A COMPLETER */) printf("r");
    else printf("-");
  if (/* A COMPLETER */) printf("w");
    else printf("-");
  if (/* A COMPLETER */) printf("x");
    else printf("-");
  printf("\n");
 }
```

```
int main(int argc, char **argv)
 {
  /* Affiche les droits en lecture seules */
  afficheDroit(/* A COMPLETER */);

  /* Affiche les droits en lecture et ex\'ecution */
  afficheDroit(/* A COMPLETER */);

  /* Affiche les droits pass\'es en param\'etres */
  if (argc > 1) afficheDroit(atoi(argv[1]));
 }
```

Les programmes que nous allons écrire utilisent des ressources communes (sémaphores, mémoire partagée, ..) à la différence des ressources habituelles (variables) qui sont locales au programme. Pour partager ces ressources (par exemple un sémaphore) entre plusieurs processus il faut pouvoir les identifier. Il faut donc être capable de générer un ensemble d'identifiants qui seront uniques pour le système et partageables entre plusieurs processus. Pour cela nous allons utiliser la fonction `ftok()`. Cette fontion permet de créer une clé à partir de deux éléments : (1) un nom de chemin dans le système de fichier, par exemple le nom du fichier source et (2) un nom de ressource (par exemple un sémaphore) identifiée par un caractère fixé par l'utilisateur. Le point 1 permet de s'assurer la clé n'est pas utilisée par d'autres programmes, le point 2 permet de distinguer différente ressources.

Exercice 4. Ecrire deux programmes `ftok1.c` et `ftok2.c` qui créent chacun une clé unique pour des objets représentés par les caractères 'A' et 'B'. Que se passe-t-il si l'on exécute deux fois ce programme ? Que se passe-t-il si deux utilisateurs différents utilisent ce programme ? Ecrire un programme `ftok3.c` qui crée une clé qui dépend de l'utilisateur qui exécute le programme (cf. la fonction `getenv()`).

2.2. **Les sémaphores.** Pour créer ou obtenir la référence d'un sémaphore en C on utilise la fonction `semget()`. Le système retourne un identificateur auquel sont attachés n sémaphores (on crée en fait un groupe de sémaphores). A chaque identificateur sont associés des paramètres (`semflag`), il s'agit en particulier d'une combinaison des droits (comme pour les fichiers). Pour créer un groupe de sémaphores l'utilisateur doit fournir une clé unique, le nombre de sémaphores dans le groupe et les paramètres. Associées (cf. exercice préliminaire) à ces paramètres des constantes permettent de définir si l'on souhaite créer le sémaphores s'il n'existe pas (`IPC_CREAT`) (`semget()` peut donc aussi être utilisé pour obtenir l'identifiant d'un sémaphore existant) ou de provoquer une erreur si le sémaphore existe déjà (`IPC_EXCL`).

```
int semget(key, nsems, semflg)
key_t key ; /* Cl\'e de l'ensemble */
int nsems /* nombre de semaphores */, semflg /* Droits et Options */;
```

Exercice 5. Ecrire un programme `cree4sem.c` qui crée un ensemble de quatre sémaphores. On s'assurera que la clé est unique sur le système en utilisant la fonction `ftok()`. On s'assurera qu'aucun groupe de sémaphores n'est associé à la clé. Exécuter le programme et vérifier la création de l'ensemble avec la commande `ipcs`. Exécuter le programme une seconde fois. Remettre le système dans son état initial en utilisant la commande `ipcrm`.

L'accès et la modification des valeurs des sémaphores se fait avec la fonction `semctl()`. Elle a besoin de quatre arguments : un identificateur de l'ensemble de sémaphores (`semid`) retourné par `semget()`, le numéro du sémaphore (dans le groupe) à examiner ou à changer (`semnum`), un paramètre de commande (`cmd`) (cf. `man semctl`). Les options (qui dépendent de la commande appliquée au sémaphore) sont passées via un paramètre de type union (`arg`).

```
int semctl(semid, semnum, cmd, arg) ;
int semid, semnum, cmd ;
union semun {
  int val;                    /* value for SETVAL */
  struct semid_ds *buf;       /* buffer for IPC_STAT, IPC_SET */
  unsigned short int array[];  /* array for GETALL, SETALL */
     /* tableau de taille egale au nombre  de semaphores de l'ensemble */
  struct seminfo *__buf;      /* buffer for IPC_INFO */
} arg;
```

Exercice 6. En supposant qu'un sémaphore a déjà été crée (par l'exécution de l'exercice précédent). Ecrire un programme `initsem.c` qui met le troisième sémaphore de l'ensemble à 1, affiche la valeur du troisième sémaphore, affiche le `pid` du processus qui a effectué la dernière modification et finalement détruit l'ensemble de sémaphores.

L'appel système `semop()` permet d'effectuer des opérations sur les ensembles de sémaphores. Nous rappelons que la particularité d'unix est que **toutes** les opérations du groupe doivent être réalisables en même temps pour être effectuées (notion de groupe de sémaphores). Il utilise trois arguments : un identificateur d'ensemble de sémaphores (`semid`), un pointeur vers un tableau de structures (qui contient les opérations à réaliser) de type `struct sembuf` (`sops`), et un entier donnant le nombre d'éléments de ce tableau (`nsops`). La structure `sembuf` spécifie le numéro du sémaphore qui sera traité, l'opération qui sera réalisée sur ce sémaphore, et les drapeaux de contrôle de l'opération. Le type d'opération (P ou V avec une valeur)dépend de la valeur de `sem_op` :

- Si `sem_op` ¡ 0 /* demande de ressource */
  - si `semval` $\geq$ |`sem_op`| alors `semval` = `semval - |sem_op|` : décrémentation du sémaphore
  - si `semval` ¡ |`sem_op`| alors le processus se bloque jusqu'à ce que `semval` $\leq$ |`sem_op`|
- Si `sem_op` = 0
  - si `semval` = 0 alors l'appel retourne
  - si `semval` ¡¿ 0 alors le processus se bloque jusqu'à ce que `semval` = 0
- Si `sem_op` ¿ 0 /* restitution de ressource */ : alors `semval` = `semval` + `sem_op`

System V propose une implantation plus complète en permettant notament que l'appel devienne non-bloquant. Cela permet d'utiliser un sémaphore pour tester la disponibilité d'une ressource sans bloquer le programme.

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

struct sembuf {
short sem_num;  /* Num\'ero du s\'emaphore (0=premier)      */
         short sem_op;   /* Op\'eration sur le s\'emaphore */
         short sem_flg;  /* Options pour l'op\'eration    */
};
```

Exercice 7. Ecrire un programme `ressource.c` qui crée un ensemble de sémaphores, fixe la valeur de l'un d'eux à 1, puis demande une ressource et se met en attente pendant 10 secondes puis libère la ressource. Ecrire un second programme `ressource-test.c` qui récupère l'identificateur de l'ensemble précédent de sémaphores puis demande également la ressource attend et la libère.

Modifier ensuite le programme `ressource-test.c` pour qu'il ne soit pas bloqué mais qu'il affiche chaque seconde que la ressource n'est pas disponible.

Exercice 8. L'objectif de cet exercice est de simuler le fonctionnement d'une mine d'or. Cet mine est composée de mineurs, de pioches, de pelles et d'une seule caisse pour ranger l'or. Chaque mineur agit de la façon suivante :

(1) Il prend une pelle **et** une pioche.

(2) Il travaille entre une et cinq heures (des secondes dans la simulation) et extrait entre 1 et 1000 grammes d'or.

(3) Il rend sa pelle et sa pioche.

(4) Il ajoute l'or qu'il a extrait à la caisse.

Plusieurs mineurs peuvent travailler en même temps.

Dans cet exercice nous ne simulerons pas le remplissage de la caisse d'or.

Ecrire un programme C qui simule le fonctionnement de la mine **sans la gestion de la caisse d'or**. Votre programme devra créer autant de processus fils qu'il y a de mineurs. Rappel : Pour que le père attende la fin de ses fils, vous utiliserez le fait que la fonction `waitpid(-1,NULL,0)` retourne -1 quand un processus n'a plus de fils.

Voici un exemple d'execution pour 4 mineurs, 1 pelle et 1 pioche :

```
#./mineOr 2 1 1
cr\'eation du mineur 1 (1656)
Le mineur 1 attend 2 heures avant de travailler.
cr\'eation du mineur 2 (1657)
Le mineur 2 attend 1 heures avant de travailler.
Le mineur 2 prend une pelle et une pioche
Le mineur 2 commence \‘a travailler pour 4 heures.
Le mineur 2 a extrait 139 g d'or
Le mineur 2 rend une pelle et une pioche
Le mineur 1 prend une pelle et une pioche
Le mineur 1 commence \‘a travailler pour 2 heures.
Le mineur de pid 1657 a fini.
Le mineur 1 a extrait 248 g d'or
Le mineur 1 rend une pelle et une pioche
Le mineur de pid 1656 a fini.
Fin du travail des mineurs.
```

2.3. **La mémoire partagée.** Le partage de mémoire entre deux ou plusieurs processus (exécutant des programmes) constitue le moyen le plus rapide d'échange de données. La zone de mémoire partagée (appelée segment de mémoire partagée) est utilisée par chacun des processus comme si elle faisait partie de chaque programme. Le partage permet aux processus d'accéder à un espace d'adressage commun en mémoire virtuelle.

La création d'un segment de mémoire se fait de mamière similaire à un sémaphore avec la fonction `shmget()`

Exercice 9. Ecrire un programme `testshm.c` qui crée un segment de mémoire partagée associé à la clé 123 et affiche l'identificateur du segment et la clé unique. Relancer le programme, vérifier l'état du système avec la commande `ipcs`, remettre le système dans son état initial.

La primitive `shmctl()` est utilisée pour examiner et modifier les informations dans le segment de mémoire partagée. Elle prend trois arguments : un identificateur du segment de mémoire partagée (`shmid`), un paramètre de commande (`cmd`), et un pointeur vers une structure de type `shmid_ds` (`buf`).

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

```
struct shmid_ds {
/* Permissions d'acc\‘es       */
  struct          ipc_perm shm_perm;
/* Taille segment en octets  */
  int             shm_segsz;
/* Heure dernier attachement */
  time_t          shm_atime;
/* Heure dernier d\'etachement */
  time_t          shm_dtime;
/* Heure dernier changement  */
  time_t          shm_ctime;
/* PID du cr\'eateur           */
  unsigned short  shm_cpid;
/* PID du dernier op\'erateur  */
  unsigned short  shm_lpid;
/* Nombre d'attachements     */
  short           shm_nattch;
/* -- Les champs suivants sont priv\'es -- */
/* Taille segment en pages   */
  unsigned short  shm_npages;
/* Taille d'une page (?)     */
  unsigned long   *shm_pages;
/* Descript. attachements    */
  struct shm_desc *attaches;
                  };
```

Exercice 10. En supposant qu'un segment de mémoire partagée a été crée par le programme de l'exercice précédent. Ecrire un programme `updateShm.c` qui récupère l'identifiant du segment de mémoire partagée, affiche les informations suivantes : `uid` et `gid` du proprietaire, l'`uid` et le `gid` du créateur, le mode d'accès, la taille de la zone mémoire, le `pid` du créateur et le `pid` du processus ayant fait la dernière opération et libère la mémoire partagée.

Afin d'utiliser un segment de mémoire de mémoire partagée un processus doit s'attacher à celui-ci avec `shmat()`, cette fonction retourne un pointeur sur caractère qui est vu comme un pointeur sur une zone mémoire locale au processus. Lorqu'il a finit il doit se détacher avec `shmdt()`.

```
char *shmat(shmid, shmaddr, shmflg)
int shmid ;
char *shmaddr ;
int shmflg ;
```

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

Exercice 11. Nous allons nous interesser au problème classique du producteur-consommateur (client/serveur). Ecrire deux programmes : le premier `shm-ecrit` s'attache à un segment de mémoire déjà crée et écrit un nombre aléatoire à l'intérieur, le second `shm-lit` se connecte au même segment et affiche son pid et le nombre lu, puis le premier programme place un autre entier aléatoire... Que se passe-t-il si le second programme est exécuté plusieurs fois.

Exercice 12. Dans un exercice précédent nous avons modélisé une mine d'or avec des processus parralèles, nous allons y ajouter la gestion de la caisse d'or.

La gestion de la caisse d'or (représentée par un entier) est faite en utilisant la mémoire partagée. Modifier le programme précédent pour que le processus père crée un segment de mémoire partagée de 4 octets. Chaque mineur pourra alors ajouter la quantité d'or extrait à l'or déjà présent dans la caisse.

Attention, il faudra vous assurer qu'un seul mineur ne met à jour la caisse en même temps.

A la fin de l'execution, le processus père devra afficher le poids total de l'or contenu dans la caisse.

Voici un exemple d'execution pour 2 mineurs, 1 pelle et 1 pioche :

```
./mineOr 2 1 1
cr\'eation du mineur 1 (1656)
Le mineur 1 attend 2 heures avant de travailler.
cr\'eation du mineur 2 (1657)
Le mineur 2 attend 1 heures avant de travailler.
Le mineur 2 prend une pelle et une pioche
Le mineur 2 commence \‘a travailler pour 4 heures.
Le mineur 2 a extrait 139 g d'or
Le mineur 2 rend une pelle et une pioche
Le mineur 2 met 139 g d'or dans la caisse
Le mineur 1 prend une pelle et une pioche
Le mineur 1 commence \‘a travailler pour 2 heures.
Le mineur de pid 1657 a fini.
Le mineur 1 a extrait 248 g d'or
Le mineur 1 rend une pelle et une pioche
Le mineur 1 met 248 g d'or dans la caisse
Le mineur de pid 1656 a fini.
Fin du travail des mineurs, la caisse contient: 387 g d'or.
```

2.4. **Application : Une bataille navale en mémoire partagée.** Lors de ce TP nous avons vu les notions de mémoire partagée et de vérification du principe d'exclusion mutuelle à l'aide de sémaphores. Pour appliquer cela, nous allons réaliser un jeu de bataille navale. Chaque joueur utilise un processus, la communication est réalisée à travers un segment de mémoire partagée dans lequel chaque joueur écrit à tour de rôle le résultat du coup précédent et son tir.

Exercice 13. Ecrire un jeu de bataille navale en mémoire partagée pour deux joueurs. Chaque joueur lancera une instance du programme, un paramètre sur la ligne de commande indiquera s'il s'agit du `joueur1` ou du `joueur2`.Le squelette des sources et des explications complémentaires (dans le squelettes) se trouvent à l'adresse :
`http ://sis.univ-tln.fr/~bruno`.