USTHB Master 1 RSD
Département Informatique 2022/2023

TP 1 Système d'exploitation:
Outils de Communication Inter-Processus
Sémaphores + Mémoire partagée

Documentation

Introduction:

Il existe plusieurs moyens pour faire communiquer deux processus s'exécutant sur la même machine sous linux.

- Les pipes ou tubes: communication entre processus père et processus fils a travers des descripteurs de fichiers (communication entre processus liés).
- Les tubes nommés (named pipes) ou les FIFO: ce sont des tubes ayant une existence dans le système de fichiers (un chemin d'accès). Communication entre processus non liés.

Ces deux moyens sont gérés comme des entrées sorties et donc sont couteux en temps d'exécution. Il existe aussi des moyens de communication gérés directement par le noyau Linux et qu'on regroupe sous l'acronyme IPC System V (IPC : Inter-Process Communication, System V est le noyau Unix ancêtre de Linux). Il y a 3 sortes d' IPCs

- Les files de messages (message queues)
- Les segments de mémoire partagée (shared memory segments)
- Les semaphores Linux

Ce TP va aborder les IPCs System V plus particulièrement les sémaphores et la mémoire partagée.

I- Identification unique des ressources IPC quel que soit leur type

Les ressources IPC sont identifiées par une clé unique dans tout le système. Avant de créer une ressource IPC, on doit donc fournir une clé unique qu'on peut obtenir avec la fonction **ftok** (file to key) qui permet de créer une clé a partir d'un chemin valide et d'une lettre.

key_t key = ftok (char* pathname, char project)
Exp: key_t key =ftok ("/home/etudiant/", 'c')

Un autre identifiant est généré par le système lors de la création d'une ressource IPC d'un type donné et qui permet d'identifier les ressources de même type.

II- Commandes Linux pour lister et manipuler les IPCs en ligne de commande:

Les IPCs System V ont la particularité de rester en mémoire même après que le processus les ayant créés ait terminé son exécution (si ce dernier n'a pas supprimé les ressources qu'il a crée). Pour créer

lister, supprimer des IPCs en ligne de commande on utilise les commandes suivantes:

Création d'une ressource IPC	ipcmk (options: -s sémaphores, -m mémoire, -q fille de messages)
Lister toutes les ressources IPC existantes dans le systeme	# ipcs -a (ou -s, -m, -q pour lister chaque type a part)
Lister les details d'une ressource IPC	# ipcs -s/-m/-q -i id Exp. # ipcs -s -i 32768
Suppression d'une ressource IPC	#ipcrm -s/-m/-q id Exp. suppression du semaphore d'id. 6766: # ipcrm -s 6766
Obtenir les limites d'une ressource IPC	# ipcs -m/-s/-q -l
Le dernier processus qui a accédé a la ressource (option -p de ipcs)	# ipcs -m -p

III- Création d'un groupe de sémaphores: la fonction semget

- Inclure le fichier entête #include <sys/sem.h>

Les sémaphores system V sont créés par groupes de **n** sémaphores. Un sémaphore est une structure **semid_ds** qui contient toutes les informations du groupe. Les sémaphores d'un même groupe sont numérotés de **0 a n**.

La fonction **semget (key, n, flags)** permet de **créer** ou **obtenir** l'identifiant d'un groupe de sémaphores existants. **N** est le nombre de sémaphores dans le groupe, flags est une combinaison (ou logique) d'options de création. L'option **IPC_CREAT** permet de créer un nouveau groupe de sémaphores, l'option **IPC_EXCL** permet de forcer un retour a **-1** si le groupe existe déjà. La 3eme option nécessaire est les droits d'accès au nouveau groupe de sems (0666 pour tous les droits).

Exp: int semid=semget(key, n, IPC_CREAT| IPC_EXCL| 0666)

- L'appel **semget (key, n, 0)** retourne l'identifiant du groupe de sémaphores existant ayant la cle **key**.

- **Le retour de semget:** -1 erreur sémaphore existant si IPC_EXCL est indiquée dans le flag ,
  0 indique aussi un sémaphore existant si IPC_EXCL n'est pas utilisée,
  n > 0 l'identifiant du nouveau groupe de sémaphores.

IV- Manipulation de semaphores: la fonction semctl

La fonction **int semctl (semid, semnum, command, arg)** permet d'initialiser la valeur d'un sémaphore, récupérer la valeur, supprimer un groupe de sémaphores etc. La liste complète des **commandes** possible est disponible dans le manuel (man 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). Les options (qui dépendent de la commande appliquée au sémaphore) sont passées via un paramètre de type union semnum (arg).

- Initialisation d'un sémaphore: on utilise la fonction **semctl** avec la commande **SETVAL**. **arg** a une interprétation différente selon la commande utilisée. Dans le cas de **SETVAL**, **arg** sera de type **int** et prendra la valeur que l'on veut affecter à notre sémaphore.

- Suppression d'un groupe de semaphore:
Pour supprimer un groupe de sémaphores existant on utilise la fonction **semctl** avec la commande IPC_RMID: semctl (semid, 0, IPC_RMID,0) le deuxième argument est ignoré ici.

IV - Opérations P( ) et V( ): La fonction semop

Les opérations P et V permettent respectivement de décrémenter (avant SC) et incrémenter la valeur d'un sémaphore (après SC). Une troisième opération est possible aussi Z et qui permet de bloquer un processus jusqu'a ce que la valeur d'un sémaphore atteint 0. L'incrémentation et décrémentation de la valeur d'un sémaphore est modélisée dans une structure **sembuf** définie comme suit :

```
struct sembuf { short sem_num; short sem_op; short sem_flg; };
```

sem_num est le numéro du sémaphore à utiliser, **sem_op** est l'opération à réaliser et **sem_flg** contient les paramètres de l'opération (dans notre cas, il n'est pas important, on le met donc à 0).

sem_op prend les valeurs possibles suivantes:

- **Supérieure à zéro (V)** : La valeur du sémaphore est augmentée de la valeur correspondante. Tous les processus en attente d'une augmentation du sémaphore sont réveillés.
- **Egale à zéro (Z)** : Teste si le sémaphore a la valeur 0. Si ce n'est pas le cas, le processus est mis en attente de la mise à zéro du sémaphore.
- **Inférieur à zéro (P)** : La valeur (absolue) est retranchée du sémaphore. Si le résultat est nul, tous les processus en attente de cet événement sont réveillés. Si le résultat est négatif, le processus est mis en attente d'une augmentation du sémaphore.

Une fois, la structure bien remplit par l'opération, elle sera transmise à la fonction semop :

```
int semop(int semid, struct sembuf *ops, unsigned nbops)
```

nbops est le nombre d'opérations à exécuter c'est à dire le nombre de sémaphores à manipuler dans le groupe identifie par **semid**. Qui est aussi égal au nombre de structures sembuf passées grâce au pointeur *ops.

V-La mémoire partagée: Les fonction shmget et shmat

Utiliser le manuel pour avoir la signature des deux fonctions : man 2 shmget, man 2 shmat

Le mécanisme de communication inter-process par mémoire partagée fonctionne de la même manière que les sémaphores.
Une fonction **shmget** (shared memory get) permet d'allouer un espace mémoire partagée en faisant appel au noyau. On doit lui fournir une clé unique obtenue avec la fonction ftok () et spécifier la taille de la zone à allouer. Si l'opération est réussie, **shmget** retournera un identifiant de la zone à utiliser par tous les processus souhaitant se partager la zone.

Un seul processus fait appel à **shmget** tandis que tout processus partageant l'accès à la même zone exécutera la fonction **shmat** (shared memory attache). Le code suivant illustre un exemple d'utilisation de ces fonctions. On souhaite créer une zone partagée contenant une structure sdata déclarée comme suit.

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

typedef struct data{
int a;
int b;
int tab[10];
}sdata;

int main(){
//creation du segment a faire dans un seul processus puis passer
l'identifiant aux autres

key_t key=ftok("/Users/user",5); //modifier le chemin

int shmid=shmget(key,sizeof(sdata),IPC_CREAT|IPC_EXCL|0666);

if(shmid==-1){    //la zone existe deja !
shmid=shmget(key,sizeof(sdata),0);  //recuperer son id

printf("Segment existe deja d'id:%d\n",shmid);
}else  printf("Segment mémoire d'id:%d\n",shmid);

//tout les processus doivent appeler shmat pour attacher une adresse a la
zone memoire le //pointeur permet par la suite d'ecrire directement des
donnes dans la zone partagée.

sdata *sd=NULL:
sd=shmat(shmid,sd,0);

//maintenant on peut écrire dans la zone via la structure.
sd->a=10;
sd->b=5;

 printf("val écrites:%d, %d\n",sd->a,sd->b);
return 0;
}
```

La série d'exercices

Consignes générales pour ce TP:

Les exercices d'initiation aux sémaphores 1, 2 et 3 doivent aboutir à une librairie de fonctions pour la création, destruction et manipulation de sémaphores. Exemple, créer un fichier semaphores2.h (semaphores.h existe déjà) qui va contenir les fonctions suivantes: semcreate (exercice 1), seminit, semdestroy, P, V, Z.

Le rapport : Rapport du TP à rendre contenant la bibliothèque de fonctions semaphores2.h et les exercices 3, 4 et 5 et 6.

Exercise 1:

1- Ecrire une fonction **int semcreate (key_t cle, int n)** qui crée un ensemble de **n** sémaphores (appel à la fonction **semget**). Utiliser la fonction **ftok** pour obtenir une clé unique a associer au groupe de sémaphores. Vérifier qu'aucun groupe de sémaphores existant n'est associé à la même clé (le retour -1 de **semget**!). Si le groupe existe déjà, on récupère son identifiant (avec un deuxième appel à **semget**) et on le retourne.

2- Ecrire un programme qui teste la fonction **semcreate** avec n=4. Utiliser la commande **ipcs** pour visualiser le groupe de sémaphores crées (voir tableau section II).

Exercise 2: *traiter les questions dans l'ordre jusqu'à la question 4, lire la page 4 tester le programme de la mémoire partagée puis continuer vers les questions 5 et 6*

1- Ecrire une fonction **seminit (int idsem, int numsem, int initval)** qui permet d'initialiser la valeur du sémaphore numéro **numsem** (du groupe **idsem**) avec la valeur **initval.**

2. Ecrire un programme qui permet de tester la fonction **seminit,** en travaillant sur le groupe de sémaphores qui a déjà été créé dans l'exercice précédent. Initialisez le troisième sémaphore de ce groupe a 1, afficher sa valeur, afficher le **pid** du processus qui a effectué la dernière modification sur ce sémaphore.

3. Modifier le programme pour initialiser tous les sémaphores du groupe a 1 puis afficher leurs valeurs (avec un seul appel à semctl, en utilisant les commandes SETALL et GETALL).

4. Ecrire les fonctions P(int semid, int numsem) et V(int semid, int numsem), Z(int semid, int numsem) permettant de réaliser les 3 opérations sur le sémaphore numsem.

5-Implémenter une fonction **barriere** réutilisable. **Rappel principe de la barrière** : n processus sont lancés en parallèles et doivent être tous au niveau de la barriere pour continuer leur exécution ensemble. Chaque processus qui exécute la barriere, il incrémente un compteur partagé initialement à zero et teste si tout le monde est déjà la, sinon il se bloque (dans file d'attente d'un séma) et attend d'être libéré par le dernier processus qui arrive à la barriere (celui-ci va libérer tout le monde)**.NB**. Ici il faut utiliser un segment de mémoire partagé (voir page 4) pour le compteur, et attention toute manipulation de ce compteur doit se faire en exclusion mutuelle).

6. Détruire le groupe de sémaphores à la fin du programme test en faisant appel à une fonction semdestroy(int semid) (que vous devez implémenter).

Exercice 3: (un premier exemple d'application)

On souhaite simuler l'accès concurrent à une seule imprimante par plusieurs processus. Un seul processus utilise l'imprimante à un instant donné. Écrivez un programme Spool.c qui se duplique pour créer N fils (La valeur de N est transmise par clavier). Chaque fils tente d'accéder à la ressource : une fois dedans, il dort un temps aléatoire entre 1 et 3 secondes, affiche un message indiquant qu'il a terminé d'utiliser la ressource ainsi que la durée d'utilisation, et ensuite il libère la ressource. Le père attend la terminaison de tous ses fils avant d'indiquer sa terminaison. Que se passera –t-il si un des processus se termine accidentellement pendant qu'il est en train d'utiliser la ressource ? testez cette situation en envoyant signal 9 au processus. Vérifiez l'état du sémaphore utilisé avec la commande ipcs. Pour régler le problème rencontré utiliser les options disponibles pour l'opération P (voir man 2 semop lire la partie qui explique les options possibles et tester).

Exercice 4: Producteur consommateur avec sémaphores+ mémoire partagée (voir doc page 4 pour la mémoire partagée)

Un processus producteur et un processus consommateur se partagent l'accès à une zone mémoire permettant de stoker 10 valeurs entières (un tableau) que le producteur dépose et le consommateur récupère pour les afficher (une seule valeur à la fois). Ce tableau est géré de manière circulaire. Un producteur écrit toujours à la prochaine case vide et un consommateur consomme toujours la prochaine case pleine du tableau (utilisation de deux indexes **indxl** et **indxr** initialisés à zéro).

1- Ecrire deux programmes **producteur.c** et **consomateur.c** qui vont écrire et lire dans une Zone de mémoire partagée qu'on va appeler data (une structure : int tab[10], int indxl, indxr), préalablement crée par un autre programme **creat.c** (qui crée aussi les sémaphores nécessaires).

2- Tester en lançant d'abord le programme **creat** puis lancer producteur et consommateur dans deux fenêtres différentes. Chaque processus doit afficher les éléments qu'il dépose ou qu'il consomme (selon le cas). Ajouter des sleeps pour ralentir l'exécution. Qu'est ce qui se passe si on arrête un des processus ? Vérifier aussi l'état des sémaphores pendant cet arrêt (à partir d'une autre fenêtre du terminal utiliser la commande ipcs –s –i).

3- Généraliser la solution à n consommateurs et n producteurs (ajouter deux sémaphores binaires pour réaliser l'exclusion mutuelle entre les producteurs et entre les consommateurs). Tester avec 2 producteurs et deux consommateurs.

Exercice 5 : H2O

Dans ce problème de synchronisation il s'agit de simuler la formation de molécules H2O dans la nature en utilisant deux types de processus: **processus oxygen** et **processus hydrogen** (deux programmes **oxygen.c** et **hydrogen.c**) qui s'exécutent tous en parallèles. Le nombre exact de

processus de chaque type est un paramètre à fixer. Pour créer ces processus on utilise la fonction fork (créer des fils au niveau de chacun des programmes).

Dans le but d'assembler ces processus en une molécule H2O il faut utiliser une barrière de synchronisation pour 3 processus exactement : 1 oxygen et deux hydrogen avec les règles suivantes :

- Si un processus oxygen arrive à la barrière en premier il doit attendre l'arrivée de deux processus hydrogen pour former la molécule.
- Si un procesus hydrogen arrive en premier à la barrière il doit attendre l'arrivée d'un processus oxygen et de deux processus hydrogen au niveau de la barrière.
- Une fois les 3 threads passent la barrière, tous les trois vont appeler une fonction bondOxygen() ou bondHydrogen() selon la cas, la fonction bondOxygen() va simplement afficher un message : molécule H2O formée avec le PID du processus oxygen, bondHydrogen() va afficher le pid du processus hydrogen.
- Il faut s'assurer que les 3 processus de la molécule courante ont fini d'appeler les fonctions bondOxygen et bondHydrogen avant que les processus de la molécule suivante ne le fassent (section critique). Donc on libere le mutex qu'à la fin du processus oxygen (puisqu'il y a un seul oxy et deux hydro, le oxygen sera responsable de la libération du mutex).

Exercice 6: Un outil de monitoring de sémaphores

Ecrire un programme sem-monitor.c qui donnera la même sortie que la commande:
ipcs -s -i 32768 ici elle permet de visualiser les détails d'un groupe de sémaphores semid= 32768. On voudrait que l'affichage se rafraichisse automatiquement à chaque modification survenue sur les sémaphores du groupe. Le programme doit fonctionner comme suit:
./sem-monitor semid.
Tester ce programme avec le programme crée dans l'exercice 3 (lancés en même temps dans deux fenêtres du terminal).