Programmation Concurrente

TD – IPC Unix

0. Les commandes shell ipc

Les IPC (Inter Process Communication) sont toutefois très différents des tubes/pipes car ils sont totalement détachés du système de fichier, ce qui est très rare dans le monde Unix. Ils permettent de faire communiquer des processus *sans lien de parenté* mais *situés sur la même machine*. Deux fonctionnalités de communication sont fournies :

1. Les segments de mémoire partagée, orientées, comme leur nom l'indique, vers le partage d'informations communes
2. Des boîtes aux lettres, où les processus s'envoient des messages

Afin de sécuriser l'accès aux segments de mémoire partagée, mais également à toute ressource critique, un mécanisme de synchronisation orienté *sémaphores* est aussi proposé.

Toutes les fonctionnalités IPC sont créées à l'aide d'une clef numérique. Accéder à une IPC suppose, soit de connaître son identificateur, soit de disposer de la clef numérique qui permet de récupérer l'identificateur. Ce système n'est pas très heureux car le créateur est obligé de rendre publique d'une manière ou d'une autre les clefs qu'il désire partager.

Deux commandes shell particulièrement importantes sont associées aux IPC.

- *ipcs :* Cette commande permet d'afficher la liste des fonctionnalités IPC présentes sur le système.
  - `ipcs [-s] [-m] [-q]`
  - `-q` pour afficher les boîtes aux lettres
  - `-m` pour afficher les segments de mémoire partagée
  - `-s` pour afficher les ensembles de sémaphore
- *ipcrm :* Utilisée pour supprimer une fonctionnalité qui a été oubliée par les programmes qui l'utilisaient
  - `ipcrm [-q|-Q|-m|-M|-s|-S] nombre`
  - *-q Boîte aux lettres par id*
  - *-Q Boîte aux lettres par clef*
  - *-m Mémoire partagée par id*
  - *-M Mémoire partagée par clef*
  - *-s Sémaphores par id*
  - *-S Sémaphores par clef*

Il est très important de détruire les fonctionnalités inutilisées car elles survivent indépendamment des processus. Si votre processus plante avant d'avoir effectué son ménage, il faut alors utiliser `ipcs` et `ipcrm` pour les détruire.

Pour détruire si nécessaire tous les ipc après la terminaison de vos processus, je vous propose de mettre dans un fichier ipcrm.awk le code suivant :

```
BEGIN{
  commande=""
  "whoami" | getline utilisateur
}
$5 == utilisateur{
  commande=commande " -"$1 " "$2
}
END{
  system("ipcrm "commande)
}
```

Reste à détruire le ipc créé en appelant la commande :

```
ipcs | awk -f ipcrm.awk
```

1. Les segments de mémoire partagées

Les segments de mémoire partagée permettent aux processus de partager de l'information, directement et sans contrôle. Ils permettent d'allouer une zone de centrale qui sera accessible directement par les processus connaissant la *clé* de cette zone, au travers d'un pointeur. Quatre primitives fondamentales permettent de manipuler les segments de mémoire partagée :

```
#include <sys/shm.h>
int shmget(key_t cle , size_t taille , int flag );
void * shmat ( int identificateur , NULL, int option );
int shmdt ( const void * adresse );
int shmctl ( int identificateur , int operation , NULL);
```

1. `shmget(cle,taille,flag)` retourne l'identificateur d'un segment à partir de sa clé (`cle`) ou -1 en cas d'échec. Le segment sera créé s'il n'existait pas encore. On peut utiliser la clé `IPC_PRIVATE` pour la création quand il n'est pas utile ensuite d'acquérir l'identificateur. Le paramètre `taille` donne le nombre d'octets du segment (s'il a déjà été créé, la taille doit être inférieure ou égale à la taille de création). Le paramètre `option` est une combinaison (par OU bit à bit) de constantes (telles que `IPC_CREAT` pour la création) et de droits d'accès (comme `0666`). Par exemple pour créer un segment on utilisera typiquement l'option `IPC_CREAT|0666`, et pour l'acquisition simplement `0666`.

2. `shmat(identificateur,NULL,option)` sert à *attacher* un segment, c'est à dire à obte- nir une fois que l'on connaît son identificateur, un pointeur vers la zone de mémoire par- tagée. L'option sera `SHM_RDONLY` pour un segment en lecture seule ou `0` pour un segment en lecture/écriture. Cette primitive retourne l'adresse de la zone de mémoire partagée ou `(void *)(-1)` en cas d'échec.

3. `shmdt(adresse)` sert à *détacher* le segment attaché à l'adresse passée en paramètre. Retourne 0 en cas de succès, ou -1 en cas d'échec.

4. `shmctl(identificateur,operation,NULL)` sert au contrôle (par exemple la suppres- sion) du segment dont l'identificateur est `identificateur`. Pour supprimer le segment, la valeur du paramètre `operation` est `IPC_RMID` (la suppression ne sera effective que lorsque plus aucun processus n'attachera le segment). Retourne 0 en cas de succès, ou -1 en cas d'échec.

2. Les sémaphores

Les sémaphore se programme de manière assez similaire aux segments de mémoire partagés.

```
#include <sys/sem.h>
int semget(key_t cle , int nbSemaphores , int flag );
int semctl (int identificateur , int numeroSemaphore , int operation, NULL);
int semop(int identificateur, struct sembuf * ensembleOps, int nbOps);
```

`semget(cle, nbSemaphores, flag)` retourne l'identificateur d'un tableau de sémaphore à partir de sa clé (`cle`) ou -1 en cas d'échec. Le tableau de sémaphore sera créé s'il n'existait pas encore. Le paramètre `option` est une combinaison (par OU bit à bit) de constantes (telles que `IPC_CREAT` pour la création) et de droits d'accès (comme `0666`). Par exemple pour créer un sémaphore on utilisera typiquement l'option `IPC_CREAT|0666`, et pour l'acquisition simplement `0666`. A sa création, un sémaphore est initialisé à 0.

`Semctl` permet d'effectuer différentes opération sur un seul des éléments du tablerau de sémaphore créé :

- l'identificateur et numéroSemaphore permettent d'identifier le sémaphore concerné
- par exemple, `operation==IPC_RMID`, suppression du tableau de sémaphore

`semop` permet de réaliser les opérations *up et* down à l'aide de la structure sembuf.

```
struct sembuf
{
  unsigned short int sem_num; /* Numero du semaphore sur lequel
s'applique l'operation */
  short              sem_op;  /* Operation up, down */
  short              sem_flg; /* Options sur l'operation */
};
```

Chaque opération est ainsi indépendante et peut même avoir des options particulières. Voici quelques explications concernant chacun des champs de la structure :

sem_num
: Numéro du sémaphore sur lequel va s'appliquer l'opération. Dans un ensemble de sémaphores, les numéros commencent à 0.

sem_op
: C'est le champ qui détermine l'opération, les valeurs possibles sont les suivantes :
sem_op > 0 Opération *up*
sem_op < 0 Opération *down*

3. Génération de clés par `ftok`

La fonction `ftok` permet de générer une clé relativement simplement à partir chemin absolu `chemin_fichier` et des huit bits de poids faible de l'entier non nul `identificateur`. En cas d'erreur, cette primitive retourne −1.

```
#include <sys / types .h>
#include <sys/ipc.h>
key_t ftok(char * chemin_fichier , int identificateur );
```

4. Gestion d'un parking souterrain (version 1)

Le but de ce travail est d'implanter une simulation de gestion d'un parking souterrain. Le parking dispose d'un certain nombre de places initial. Des voitures peuvent se présenter à différentes bornes pour demander un ticket. Si au moins une place est libre, un ticket est délivré et le nombre de places libres est décrémenté. S'il n'y a plus de places disponibles, un message d'erreur est simplement délivré.

Dans le cadre de ce travail, chaque borne sera un programme distinct : il n'y aura aucun lien de parenté entre les différents processus, ils auront tous leur propre code source. Le nombre de places libres sera contenu dans un segment de mémoire partagée que connaîtront toutes les bornes. Le programme parking.c ci-après crée et initialise le segment :

```
#include <stdio .h>   // Pour printf ()
#include <stdlib .h>  // Pour exit(), NULL
#include <unistd .h>  // Pour pause ()
#include <fcntl .h>   // Pour open (), O_CREAT O_WRONLY
#include <signal .h>  // Pour signal ()
#include <sys/types.h>// Pour key_t
#include <sys/ipc.h>  // Pour ftok (), IPC_CREAT , IPC_RMID
#include <sys/shm.h>  // Pour shmget (), shmat (), shmdt (), shmctl ()

#include "def.h"      // Pour SHM_CHEMIN e t SHM_ID


int shmid;            // Identification du segment
int *shmadr;          // Adresse d'attachement du segment

/* Fonction executee a la reception du signal SIGINT */
void traitant_sigint(int numero_signal) {
       shmdt (shmadr);                    // Detachement du segment
       shmctl (shmid , IPC_RMID, NULL);   // Destruction du segment
       exit (0);
}

/* Fonction creant un segment de memoire partagee de 'taille' octets et dont * la cle est
construite a partir de 'chemin_fichier' et de 'identificateur '. * Cette fonction retourne l '
identificateur du segment.
*/
int shm_creation (char * chemin_fichier , int identificateur , int taille ) {
       int fd, shmid;
       key_t cle ;

       /* Generation de la cle a partir de 'chemin_fichier' et identificateur' */
       cle = ftok (chemin_fichier, identificateur);

       /* Creation d'un segment de memoire partagee de ' taille ' octets . */
       shmid = shmget(cle, taille, IPC_CREAT|0666);

       return shmid ;
}

int main() {
       /* Creation du segment de memoire partagee de taille un entier. */
       shmid = shm_creation (SHM_CHEMIN, SHM_ID, sizeof (int));

       /* Attachement du segment et recuperation de son adresse. */
       shmadr = (int *) shmat (shmid, NULL, 0);

       /* Initialisation a 20 de l'entier contenu dans le segment. */
       *shmadr = 20;
       Printf ("parking : identificateur=%d, places=%d.\n", shmid, *shmadr);

       /* Deroutement de SIGINT et endormissement jusqu'a reception d'un signal. */
       Signal (SIGINT, traitant_sigint);
       pause ();

       return 0;
}
```

Questions :

1. Étudiez le programme `parking` et comprenez son fonctionnement.
2. Depuis un terminal, exécutez la commande `ipcs`. Quelles informations vous donne-t-elle ?
3. Compilez et exécutez sans l'arrêter le programme `parking`. Exécutez depuis un autre terminal la commande `ipcs`. Qu'observez-vous ?

4. Arrêtez le programme `parking` par la combinaison `<ctrl-C>` (envoi du signal `SIGINT`). Exécutez depuis un autre terminal la commande `ipcs`. Qu'observez-vous ?
5. Mettez en commentaire le déroutement du signal `SIGINT` dans le programme `parking` et refaites les deux manipulations précédentes. Qu'observez-vous ? Qu'en concluez-vous ?
6. La commande permettant de détruire un segment de mémoire partagée ou un sémaphore depuis le shell est `ipcrm`. Détruisez à l'aide de cette commande le segment qui à présent ne sert plus à rien.

5. Ajout de bornes d'entrée

Les bornes de distribution des tickets du parking sont des programmes indépendants capables de lire et d'écrire dans le segment de mémoire partagée pour mettre à jour le nombre de places. Pour simplifier, on ne gèrera pas dans ce travail la sortie des voitures du parking : les voitures demandent des tickets auprès des bornes qui en délivrent tant qu'il y en a. Les bornes 1 et 2 ont des codes légèrement différents correspondant aux algorithmes suivants :

Borne entrée 1

```
tant_que vrai faire
si nombre_de_places > 0 alors
        afficher "Demande acceptée"
        dormir 2 secondes
        décrémenter nombre_de_places
        afficher " ticket", nombre_de_places
sinon
        afficher "Pas de place"
        dormir 1 seconde
```

Borne entrée 2

```
tant_que vrai faire
si nombre_de_places > 0 alors
        afficher "Demande acceptée"
        décrémenter nombre_de_places
        afficher "ticket", nombre_de_places
        dormir 1 seconde
sinon
        afficher "Pas de place"
        dormir 1 seconde
```

Implantez les programmes des deux bornes. Lancez le programme `parking` (n'oubliez pas de décommenter le déroutement de `SIGINT` et de recompiler) puis, dans deux terminaux séparés, lancez les programmes des deux bornes aussi simultanément que possible. Renouvelez l'expérience plusieurs fois.

Questions

- Que remarquez-vous ? Est-il possible que le nombre de places devienne négatif ? Pourquoi ?
- Proposez une solution utilisant un ou plusieurs sémaphores pour résoudre le problème en modifiant dans un premier temps les algorithmes proposés, puis en les implémentant à l'aide des sémaphores Unix.

6. Ajout des bornes de sortie

Complétez le code pour ajouter une ou plusieurs bornes de sorties.