# Programmation Concurrente

## TD – IPC Unix

# 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 sémaphore

# 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 ;


       /* Creation du fichier ' chemin_fichier ' pour generer la clé */
       fd = open (chemin_fichier, O_CREAT|O_WRONLY, 0644);
       close (fd);

       /* 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 1**

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

**Borne 2**

```
tant_que vrai faire
si nombre_de_places > 0 alors
        afficher "Demande acceptée"
        décrémenter nombre_de_places
        afficher "Impression ticket"
        afficher nombre_de_places
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 implémantant à l'aide des sémaphores Unix.

ATTENTION : L'implantation Unix est beaucoup plus riche qu'une simple transposition des primitives `Down` et `Up`. Elle permet entre autre l'acquisition simultanée d'exemplaires multiples de

plusieurs sémaphores différents (c'est à dire $n_1$ exemplaires d'un sémaphore $S_1$, $n_2$ exemplaires d'un sémaphore $S_2$, ...).

```
#include <sys/sem.h>
struct sembuf {
        unsigned short int sem_num; /* Numero de semaphore. */
        short sem_op ; /* Operation sur le semaphore . */
        short sem_flg ; /* Option . */
};
```

Le premier champ, `sem_num`, donne le numéro de sémaphore (les numéros commençant à 0).
Le second champ, `sem_op`, donne l'opération en elle même, son signe indique l'opération : négatif ⇒ opération *Down*(*sem_num*), positif ⇒ opération *Up*(*sem_num*).
Nous n'entrerons pas dans les détails du dernier champ.

Il y a simplement trois primitives fondamentales pour la manipulation des sémaphores Unix :

```
#include <sys/sem.h>
int semget ( key_t cle , int nb_semaphores , int options );
int semop ( int identificateur , struct sembuf * tab_op , int nb_op );
int semctl(int identificateur , int semnum, int operation );
```

1. `semget(cle,nb_semaphores,options)` sert à créer ou à acquérir l'identificateur d'un ensemble de sémaphores à partir de sa clé (`cle`). On peut utiliser la clé `IPC_PRIVATE` pour la création quand il n'est pas utile ensuite d'acquérir l'identificateur. Le paramètre `nb_semaphores` est le nombre de sémaphores de l'ensemble (s'il a déjà été créé, le nombre doit être inférieur ou égal au nombre lors de la création). Le paramètre `option` est une combinaison (par OU bit à bit) de constantes (telles que `IPC_CREAT` pour la création et `IPC_EXCL` pour renvoyer une erreur si l'ensemble existe déjà) et de droits d'accès (comme `0666`). Par exemple pour créer un ensemble on utilisera typiquement l'option `IPC_CREAT|IPC_EXCL|0666`, et pour l'acquisition simplement `0`.
2. `semop(identificateur,tab_op,nb_op)` réalise sur l'ensemble de sémaphores d'identificateur `identificateur` les `nb_op` opérations passées en argument sous la forme d'un tableau `tab_op` de `nb_op` structures de type `sembuf` **atomiquement**. Cela signifie qu'elles sont toutes réalisées ou qu'aucune ne l'est (chaque opération étant bien sûr atomique). La fonction retourne 0 en cas de succès ou -1 en cas d'échec.
3. `semctl(identificateur,semnum,operation)` sert à la gestion de l'ensemble de sémaphores d'identificateur `identificateur`. Son action et sa valeur de retour dépendent de la valeur du paramètre `operation`. Par exemple, si `operation` est `IPC_RMID`, `semctl` supprime l'ensemble de sémaphores et retourne 0 en cas de succès ou -1 en cas d'échec. Si `operation` est `GETNCNT`, `semctl` retourne le nombre de processus en attente d'augmentation du sémaphore numéro `semnum`.