IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Pratiques n° 4 : Segments de Mémoire Partagée

Nom(s) :
Groupe :
Date :

Objectifs : savoir mettre au point des applications multiprocessus dont les mécanismes de communication et de synchronisation reposent sur les segments de mémoire partagée associés aux sémaphores.

1 Avant propos

Les rappels nécessaires à la réalisation de ce TP sont fournis en notes de cours au début du TD n° 3 et 4. Bien sûr les pages de `man` sont comme toujours de précieuses alliées. Vous trouverez les codes utiles sur :

`http://www.lri.fr/~bastoul/teaching/systeme`

2 Complément de notes de cours : génération de clés par `ftok`

Afin de permettre à différents programmes d'utiliser un même segment de mémoire partagée (ou un même sémaphore), il existe un mécanisme de *clé*. Chaque programe connaissant ou pouvant générer la clé unique d'un segment (ou d'un sémaphore) est en mesure d'acquérir l'identificateur de ce segment (ou de ce sémaphore) et donc de l'utiliser. La primitive `ftok` permet de générer une clé relativement simplement à l'aide du chemin vers un fichier existant et d'un entier :

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

Cette fonction utilise l'identité du fichier indiqué par le chemin absolu `chemin_fichier` et les huit bits de poids faible de l'entier non nul `identificateur` pour retourner une clé. En cas d'erreur, cette primitive retourne -1. Ainsi il suffit à tout programme de connaître le fichier et l'identificateur pour générer la bonne clé et acquérir par la suite l'identificateur du segment souhaité par la primitive `shmget` (ou du sémaphore par `semget`), s'il en a le droit.

3 Gestion d'un parking souterrain

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 :

```c
#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 et SHM_ID

int shmid;                      // Identificateur 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 cle. */
  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;
}
```

Étudiez le programme `parking` et résumez en quelques lignes son fonctionnement.

Depuis un terminal, exécutez la commande `ipcs`. Quelles informations vous donne-t-elle ? Que déduisez-vous du volume de segments de mémoire partagée par rapport aux autres mécanismes ?

Compilez et exécutez sans l'arrêter le programme `parking`. Exécutez depuis un autre terminal la commande `ipcs`. Qu'observez-vous ?

Arrêtez le programme `parking` par la combinaison `<ctrl-C>` (envoi du signal `SIGINT`). Exécutez depuis un autre terminal la commande `ipcs`. Qu'observez-vous ?

Commentez le déroutement du signal `SIGINT` dans le programme `parking` et refaites les deux manipulations précédentes. Qu'observez-vous ? Qu'en concluez-vous ?

La commande permettant de détruire un segment de mémoire partagée ou un sémaphore depuis le shell est `ipcrm`.

Détruisez le segment qui à présent ne sert plus à rien. Quelle ligne de commande précise avez-vous utilisé ?

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.

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.

À moins que vous n'ayez le temps d'adapter ou de créer votre propre bibliothèque, on vous propose d'utiliser une bibliothèque de manipulation de sémaphores très simplifiée disposant des fonctionnalités suivantes :

```
int  easysem_create (char * chemin_fichier);
int  easysem_getid  (char * chemin_fichier);
void easysem_P      (int identificateur);
void easysem_V      (int identificateur);
void easysem_destroy(int identificateur);
```

1. `easysem_create(char * chemin_fichier)` crée un sémaphore identifié par la chaîne de caractères `chemin_fichier`, initialise son compteur à 1 et retourne son identificateur.
2. `easysem_getid(char * chemin_fichier)` renvoie l'identificateur du sémaphore ayant une clé correspondant au chemin `chemin_fichier`.
3. `easysem_P(int identificateur)` réalise l'opération `P` sur le sémaphore désigné par `identificateur`.
4. `easysem_V(int identificateur)` réalise l'opération `V` sur le sémaphore désigné par `identificateur`.
5. `easysem_destroy(int identificateur)` demande au système la destruction du sémaphore désigné par `identificateur`.

Corrigez le programme `parking` ainsi que celui des bornes à l'aide de cette bibliothèque (ou de la vôtre). **Joignez tous les codes au compte-rendu de TP.**

4 Bibliothèque pour les segments de mémoire partagée

Les primitives Unix de manipulation de segments de mémoire partagée sont relativement complexes car puissantes. Pour alléger nos programmes dans les cas simples, on veut construire une bibliothèque `easyshm` de manipulation de segments de mémoire partagée. Celle-ci disposera des fonctions suivantes :

```
int    easyshm_create (char * chemin_fichier, int taille);
int    easyshm_getid  (char * chemin_fichier, int taille);
void * easyshm_getaddr(int identificateur);
void   easyshm_destroy(int identificateur);
```

1. `easyshm_create(char * chemin_fichier, int taille)` crée un segment de mémoire partagée de `taille` octets identifié par la chaîne de caractères `chemin_fichier` et retourne son identificateur. Pour générer une clé à partir de `chemin_fichier`, on pourra faire usage de la primitive `ftok` qui génère une clé à partir du chemin absolu d'un fichier (`chemin_fichier`) et d'un nombre entier (qu'on peut pour simplifier déduire du chemin, par exemple par `(int)chemin_fichier[0]`).
2. `easyshm_getid(char * chemin_fichier, int taille)` renvoie l'identificateur du segment de taille `taille` ayant une clé correspondant au chemin `chemin_fichier`.
3. `easyshm_getaddr(int identificateur)` renvoie l'adresse où on peut accéder au segment de mémoire partagée désigné par `identificateur`.
4. `easyshm_destroy(int identificateur)` détruit le segment de mémoire partagée désigné par `identificateur`.

Implantez la bibliothèque `easyshm` et utilisez-la dans les différents programmes. **Joignez les fichiers `easyshm.h` et `easyshm.c` correspondants à votre compte-rendu.**

Commentaires personnels sur le TP (résultats attendus, difficultés, critiques etc.).