IUT Orsay

DUT Informatique
Module Système S4-L
2010 / 2011

Travaux Dirigés nº 3 : Sémaphores

Objectifs : comprendre le concept de sémaphore, son utilité pour réaliser l'exclusion mutuelle et savoir utiliser son implémentation Unix.

1 Notes de cours

1.1 Concept de sémaphore

Un sémaphore est un mécanisme de synchronisation de processus inventé par le physicien et informaticien hollandais Edsger Dijkstra en 1965. Il s'agît d'une structure de données qui comprend (1) une variable entière non négative donnant le nombre de ressources disponibles et (2) une file d'attente de processus. On manipule un sémaphore uniquement au travers de trois opérations qui ont la particularité d'être **atomiques**, c'est à dire qu'elles s'exécuteront jusqu'à terminaison sans être interrompues par un autre processus :

```
Init(semaphore sem, int nombre_de_ressources);
P(semaphore sem);
V(semaphore sem);
```

1. `Init(sem,nombre_de_ressources)` est la procédure d'initalisation du sémaphore : le nombre de ressources disponibles de `sem` est initialisé à `nombre_de_ressources` et la file de processus de `sem` est vide. Cette opération n'est effectuée qu'une et une seule fois.
2. `P(sem)` (du hollandais *Proberen* : tester, en français «***P**uis-je ?*») met le processus dans la file d'attente si le nombre de ressources du sémaphore `sem` est à 0 et décrémente le nombre de ressources sinon.
3. `V(sem)` (du hollandais *Verhogen* : incrémenter, en français «***V**as-y !*») incrémente le nombre de ressources du sémaphore `sem` ou réveille un processus s'il y en a en attente.

1.2 Implantation Unix

L'implantation Unix est beaucoup plus riche qu'une simple transposition des primitives `P` et `V`. Elle permet en effet l'acquisition simultanée d'exemplaires multiples de plusieurs ressources différentes (c'est à dire n_1 exemplaires d'une ressource R_1, n_2 exemplaires d'une ressource R_2, ..., n_p exemplaires d'une ressource R_p). De plus cette implantation permet une nouvelle opération `Z` qui permet d'attendre qu'un sémaphore devienne nul. Les opérations sont décrites à l'aide de la structure de données suivante :

```
#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 $\Rightarrow$ opération $P(sem\_num)$, positif $\Rightarrow$ opération $V(sem\_num)$, nul $\Rightarrow$ opération $Z(sem\_num)$. Nous n'entrerons pas dans les détails du dernier champ.
Il y a simplement trois primitives fondamentales pour la manipulation des sémaphores Unix :

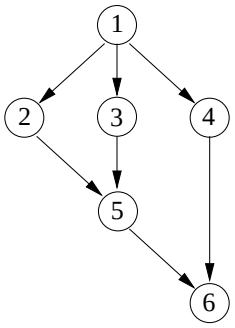
```
#include <sys/sem.h>
int semget(key_t cle, int nb_semaphores, int options);
int semop (int identificateur, struct sembuf * tab_op, int nb_op);
int semctl(int identificateur, int semnum, int operation);
```

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`.

2 Exercices

2.1 Exercice 1 : synchronisation

Soit l'ensemble de processus suivant, où les contraintes de précédence sont données par le graphe ci-dessous :



1. Donner une solution utilisant les sémaphores pour synchroniser ces processus de manière à respecter les contraintes de précédence (on ne demande pas de code C, mais un

pseudo-code pour chaque processus avec les appels aux primitives `P` et `V` nécessaires à la synchronisation ; vous préciserez combien de sémaphores vous sont nécessaires et à combien de ressources chacun est initialisé).

2. Il existe une solution n'utilisant pas plus de trois sémaphores, laquelle (si vous ne l'aviez pas trouvée à la question précédente) ?

2.2 Exercice 2 : problème des producteurs/consommateurs

Les problèmes de type producteur consommateur sont ceux dans lesquels certains processus consomment des données produites par d'autres. Ce qui rend le problème délicat, c'est que la place pour stocker les choses produites est limitée et que le nombre de processus qui produisent ou consomment n'est pas connu a priori. Par conséquent le problème introduit plusieurs types de contraintes de synchronisation :

– Un producteur doit arrêter de produire quand il n'a plus de place pour stocker ce qu'il produit.
– Un consommateur doit arrêter de consommer des choses quand l'espace est vide.
– Producteurs et consommateurs doivent éviter de se marcher sur les pieds : deux producteurs qui produisent en même temps ne doivent pas placer leur production au même endroit (en mémoire), deux consommateurs ne doivent pas consommer la même donnée, etc.

Dans cet exercice, on vous demande de compléter les trous dans les squelettes suivants.Vous disposez en tout de 3 sémaphores : `mutex`, `full` et `empty`, dont il faudra compléter l'initialisation. On suppose que les données produites sont de taille fixe, et que l'espace pour les stocker est un tampon circulaire de dimension `n`. Vous n'avez pas à vous préoccuper de l'ajout/retrait d'éléments dans le tampon, qui est réalisé automatiquement par les instruction `add` et `remove`. En revanche, invoquer `add` lorsque le buffer est plein ou `remove` lorsqu'il est vide est interdit. Attention : le nombre de lignes pointillées ne correspond pas forcément au nombre d'instructions qu'il faut utiliser !

Initialisation	Producteur	Consommateur

`Init(mutex, 1 )`
`Init(full,  0 )`
`Init(empty,...)`
	`tant_que` vrai **`faire`**
	`...`
	`P(mutex)`
	`...`
	`add(objet)`
	`...`
	`...`
	`...`
		`tant_que` vrai **`faire`**
		`...`
		`...`
		`...`
		`remove(objet)`
		`...`
		`V(empty)`
		`...`

2.3 Exercice 3 : implémentation

Implémentez les primitives de base des sémaphores définies par Dijkstra (voir notes de cours, section 1.1), `Init`, `V` et `P` à l'aide des primitives standard Unix.

3 Entraînement : exercice corrigé

3.1 Énoncé : interblocage

Un étudiant qui se spécialise en anthropologie et accessoirement en informatique s'est embarqué dans un projet de recherche pour voir s'il était possible d'enseigner les interblocages aux babouins d'Afrique. Il repère un profond canyon et y jette une corde au travers, de sorte que

les babouins puissent le traverser à bout de bras. Plusieurs babouins peuvent traverser en même temps, pourvu qu'ils aillent tous dans la même direction. Si des babouins qui se dirigent vers l'est et d'autres vers l'ouest se trouvent sur la corde au même moment, cela conduit à un interblocage (les babouins sont bloqués au point de rencontre sur la corde) : en effet, ils n'ont pas la possibilité de passer les uns par-dessus les autres alors qu'ils sont suspendus au-dessus du canyon. Si un babouin souhaite traverser le canyon, il doit vérifier qu'aucun autre babouin ne traverse en sens inverse.

1. Au moyen de sémaphores, écrivez un programme pour éviter l'interblocage. Ne traitez pas le cas d'un groupe infini de babouins se déplaçant d'un côté et interdisant tout passage à ceux qui se déplacent vers l'autre côté.
2. Reprenez la question précédente, mais en évitant la privation de ressources (*famine*). Lorsqu'un babouin qui souhaite traverser le canyon vers l'est arrive à la corde et trouve un babouin qui traverse vers l'ouest, il attend jusqu'à ce que la corde soit vide, mais aucun babouin se déplaçant vers l'ouest n'est autorisé à démarrer jusqu'à ce qu'au moins un babouin ait traversé dans l'autre sens.

3.2 Correction (essayez d'abord ! ! !)

3.2.1 Question 1

On doit ici se méfier des solutions trop simples. Pour commencer, on peut écrire un programme dans le cas simplifié où un seul babouin peut être sur la corde à un moment donné. Dans ce cas la solution est simple : la ressource est la corde et tous les babouins auront le même code.

```
P(corde)
traverser()
V(corde)
```

À présent, il s'agit de répondre à la question où plusieurs babouins peuvent avancer dans la même direction. La difficulté est que seul le premier babouin de la file doit prendre la ressource `corde`, et seul le dernier de la file, quand il a fini de traverser, doit rendre la ressource. Il faut donc un sémaphore supplémentaire pour la gestion de la corde.

Babouins de l'Est	Babouins de l'Ouest

`P(gestion_est)`
`nb_babouin_est ++`
`si` `nb_babouin_est == 1` **`alors`**
`  P(corde)`
`V(gestion_est)`
`traverser()`
`P(gestion_est)`
`nb_babouin_est --`
`si` `nb_babouin_est == 0` **`alors`**
`  V(corde)`
`V(gestion_est)`
	`P(gestion_ouest)`
	`nb_babouin_ouest ++`
	`si` `nb_babouin_ouest == 1` **`alors`**
	`  P(corde)`
	`V(gestion_ouest)`
	`traverser()`
	`P(gestion_ouest)`
	`nb_babouin_est --`
	`si` `nb_babouin_ouest == 0` **`alors`**
	`  V(corde)`
	`V(gestion_ouest)`

Il faut bien remarquer que :
– sans le sémaphore de gestion, la condition `nb_babouin_est == 1` pourrait par exemple ne jamais être vraie (deux incrémentations pourraient avoir lieu suite à un changement de processus au mauvais moment),
– il y a famine car si des babouins sont engagés, il n'y a aucune garantie pour ceux de l'autre côté de passer un jour.

3.2.2 Question 2

Pour éviter la famine, il suffit de modifier l'algorithme en ajoutant un sémaphore `ordre_arrivee` autour du protocole d'entrée. C'est donc le système qui gère la file d'attente de ceux ayant demandé la ressource qui garantira le respect de l'ordre des demandes.

Babouins de l'Est	Babouins de l'Ouest

`P(ordre_arrivee)`
`P(gestion_est)`
`nb_babouin_est ++`
`si` `nb_babouin_est == 1` **`alors`**
`  P(corde)`
`V(gestion_est)`
`V(ordre_arrivee)`
`traverser()`
`P(gestion_est)`
`nb_babouin_est --`
`si` `nb_babouin_est == 0` **`alors`**
`  V(corde)`
`V(gestion_est)`
	`P(ordre_arrivee)`
	`P(gestion_ouest)`
	`nb_babouin_ouest ++`
	`si` `nb_babouin_ouest == 1` **`alors`**
	`  P(corde)`
	`V(gestion_ouest)`
	`V(ordre_arrivee)`
	`traverser()`
	`P(gestion_ouest)`
	`nb_babouin_est --`
	`si` `nb_babouin_ouest == 0` **`alors`**
	`  V(corde)`
	`V(gestion_ouest)`