INTRODUCTION AUX SYSTEMES D'EXPLOITATION

TD2
Exclusion mutuelle / Sémaphores

S O M M A I R E

1.Généralités sur les sémaphores

1.1. Présentation

Un sémaphore est un mécanisme empêchant deux processus ou plus d'accéder simultanément à une ressource partagée. Sur les voies ferrées, un sémaphore empêche deux trains d'entrer en collision sur un tronçon de voie commun.

Sur les voies ferrées comme pour les applications que vous développez, les sémaphores ne sont qu'indicatifs :

- Si un machiniste ne voit pas le signal ou ne s'y conforme pas, le sémaphore ne pourra éviter la collision.
- De même si un processus ne teste pas un sémaphore avant d'accéder à une ressource partagée, le chaos peut en résulter.

Un sémaphore binaire n'a que deux états :

- 0 verrouillé (ou occupé),
- 1 déverrouillé (ou libre).

Un sémaphore général peut avoir un très grand nombre d'états car il s'agit d'un compteur dont la valeur initiale peut être assimilée au nombre de ressources disponibles.
Par exemple, si le sémaphore compte le nombre d'emplacements libres dans un tampon et qu'il y en ait initialement 5 on doit créer un sémaphore général dont la valeur initiale sera 5. Ce compteur :

- Décroît d'une unité quand il est pris/acquis (" verrouillé ").
- Croît d'une unité quand il est libéré (" déverrouillé ").

Quand il vaut zéro, un processus tentant de l'acquérir doit attendre qu'un autre processus ait augmenté sa valeur car il ne peut jamais devenir négatif. Le processus en attente se met donc à l'état « bloqué ».
L'accès à un sémaphore se fait généralement par deux opérations :

P : pour l'acquisition
V : pour la libération

Un moyen mnémotechnique pour mémoriser le P et le V est le suivant :

- **P**(uis-je) accéder à une ressource.
- **V**(as-y) la ressource est disponible.

Si nous devions écrire en C ces deux primitives nous aurions le résultat suivant.

P	V

void P(int* sem)
{
while (*sem <=0);
(*sem)--;
}
	void V(int *sem)
	{
	(*sem)++
	}

Les deux fonctions C précédentes pour les opérations P et V ne fonctionnent pas :

- La variable sémaphore sur laquelle pointe *sem* ne peut être partagée entre plusieurs processus, car ils ont des segments de données distinctes.
- Ces fonctions ne s'exécutent pas de manière indivisible, le noyau pouvant interrompre à tout moment un processus. On peut imaginer le scénario suivant :
  - Un processus N_1 termine la boucle *while* dans P et est interrompu avant de pouvoir décrémenter le sémaphore.
  - Un processus N_2 entre dans P, trouve que le sémaphore est égal à 1, parcourt sa boucle *while* et décrémente le sémaphore à 0.
  - Le processus N_1 reprend et décrémente le sémaphore à -1 qui est une valeur illégale.
  - Si *sem* vaut 0, P réalise une **attente active** qui n'est pas très judicieux pour utiliser au mieux l'unité centrale.

Pour toutes ces raisons, P et V ne peuvent tout simplement pas être programmées dans l'espace utilisateur.

Les sémaphores doivent être fournis par le noyau qui, lui, peut :
- Partager des données entre les processus.
- Exécuter des opérations indivisibles (ou atomiques), grâce à l'instruction en assembleur dédiée à cela (voir instruction TSL dans le cours).
- Allouer l'unité centrale à un processus prêt quand un autre processus se bloque.

1.2. Un mécanisme fourni par le noyau

La notion de sémaphore est implantée dans la plupart des systèmes d'exploitation. Il s'agit d'un concept fondamental car il permet une solution à la plupart des problèmes d'exclusion mutuelle pour tout type d'application concurrente.
Ce concept nécessite la mise en œuvre d'une variable, le sémaphore, et de deux opérations atomiques associées P et V.
Soit ***sema*** la variable, elle caractérise les ressources et permet de les gérer. Lorsqu'on désire effectuer une exclusion mutuelle entre tous les processus par exemple, il n'y a virtuellement qu'une seule ressource et on donnera à ***sema*** la valeur initiale de 1.
P(sema) correspond à une prise de ressource et V(sema) à une libération de ressource.

Lorsqu'un processus effectue l'opération P(sema) :
- si la valeur de ***sema*** est supérieure à 0, il y a alors des ressources disponibles, P(sema) décrémente ***sema*** et le processus poursuit son exécution,
- sinon ce processus sera mis dans une file d'attente à l'état « bloqué » jusqu'à la libération d'une ressource.

Lorsqu'un processus effectue l'opération V(sema) :
- si il n'y a pas de processus dans la file d'attente, V(sema) incrémente la valeur de séma,
- sinon un processus en attente est débloqué et prend le sémaphore.

2.Exercices sur les sémaphores

Dans les exercices qui suivent, nous nous concentrerons sur l'aspect algorithmique des sémaphores. Nous allons utiliser les fonctions P et V qui n'existent pas en réalité. Lors du TP, nous nous concentrerons sur la mise en œuvre (voir le cours).

2.1. Exercice N°1

Deux processus (P1 et P2, voir ci-dessous) souhaitent établir un rendez-vous avant l'exécution de la fonction RDV1() pour l'un et RDV2() pour l'autre. En utilisant les sémaphores, écrivez la séquence de pseudo-code de P1 et P2 permettant d'établir ce rendez-vous.

Décrire le comportement des deux processus à partir d'un diagramme temporel.

Aide : il s'agit ici d'établir une dépendance entre les 2 processus afin que chacun soit obligé d'attendre l'autre.

Processus 1	Processus 2

{
// rajouter votre code avant la fonction RDV1()
…
…
RDV1()
…
…
}
	{
	// rajouter votre code avant la fonction RDV2()
	…
	…
	RDV2()
	…
	…
	}

2.2. Exercice N°2

Effectuer un rendez-vous entre 3 processus P1, P2 et P3.

2.3. Exercice N°3

Généraliser le rendez-vous précédent entre N processus avec un ensemble de sémaphores initialisés à 0 : sema [i] désigne le sémaphore i.
Ecrire le code du processus Pi permettant d'établir ce rendez-vous.

2.4. Exercice N°4

On est toujours dans le cas d'un rendez-vous de N processus mais avec les contraintes suivantes :

1. Tous les processus ont le même code.
2. Il y a une variable globale qui compte le nombre de processus arrivés au rendez-vous. Soit cette variable **shm.nbArrivés** stockée dans un segment de mémoire partagée **shm** et initialement positionnée à la valeur 0.
3. C'est le dernier processus arrivé qui libère tous les autres.

La solution proposée est la suivante :

```
#define N 10 /* Nombre de processus */
int sema [N] /* un semaphore par processus initialisé à 0 au départ */
void Processus (int i)
{
      int k ;
      if (shm.nbArrivés != N-1) {
            shm.nbArrivés ++ ;
            P (sema [i]);
      }
      else {
            for (k=0 ;k<N ;k++) {
                  if (k != i) V(sema[k]) ;
            }
      }
}
```

Expliquer les principes de fonctionnement de l'algorithme proposé. La solution proposée est-elle correcte ? Sinon corrigez.

3. Lecteur/Rédacteur

Le problème de lecteur/rédacteur est un modèle d'application très utilisé dans divers domaines. Pour simplifier, on peut imaginer un ensemble de processus qui sont les rédacteurs et un autre ensemble qui sont les lecteurs. Les rédacteurs écrivent dans un fichier (ou tout autre support : un tableau, une ressource, une page web, une fenetre de chat Discord, un fichier GoogleDoc, etc) et les lecteurs y lisent. Afin que l'application soit cohérente, nous devons établir des régles de fonctionnement, par exemple qu'il n'y ait qu'un seul rédacteur qui écrive dans un fichier à un instant donné (même si plusieurs demandent à le faire), que lorsqu'un lecteur lit, aucun rédacteur ne peut écrire mais que les autre lecteurs peuvent lire aussi.
Dans cet exercice, nous allons essayer de voir comment on peut utiliser les sémaphores pour résoudre ce type de problème.

3.1. Présentation

- Soit une donnée partageable par plusieurs processus :
  - Un rédacteur qui met à jour la donnée et qui y accède en lecture/écriture.
  - Plusieurs lecteurs qui consultent la donnée en lecture uniquement.

- On souhaite obtenir un maximum de parallélisme tout en conservant une information cohérente :
  - Pas de lecteur et rédacteur en parallèle.

- Pas 2 rédacteurs en parallèle.

Le début de solution proposée est la suivante :

- Un rédacteur demande toujours l'autorisation d'utiliser la donnée.
- Un lecteur ne demande l'autorisation d'utiliser la ressource que s'il est le 1er lecteur.

Soit « ***donnee »*** le sémaphore d'accès à la donnée et dont la valeur initiale est 1. « **fichier** » est un fichier partagé entre l'ensemble des processus (on peut aussi considérer une variable partagée, mais pour la mise en œuvre on utilisera un fichier car les variables partagées n'ont pas été encore étudiées).

```
Processus REDACTEUR
Processus LECTEUR


Redacteur () {
      P(donnee) ; // je prends le sémaphore et rentre en section critique
      ecrire (fichier) ; // étant le seul à accéder à la section critique je
                           peux écrire tranquillement dans le fichier partagé
      V(donnee) ; // j'ai fini d'écrire, je rends le sémaphore pour que
                    d'autres processus puissent y lire ou y écrire
}
Lecteur () {
      if pas-de-lecteur-en-cours { //si je suis le premier lecteur, je dois
                                     demander le sémaphore
            P(donnee) ;
      }
      lire (fichier); // j'ai pris le sémaphore, je peux tranquillement lire
                        le fichier partagé (sans qu'un écrivain puisse écrire
                        en même temps). Ou sinon, je ne suis pas le premier
                       lecteur et je peux donc lire sans prendre le sémaphore
                       (car pris par le premier lecteur)
      if dernier-lecteur {
            V(donnee) ; // si je suis le dernier lecteur, je peux relâcher le
                       sémaphore pour qu'éventuellement des écrivains
                       puissent le prendre et écrire.
      }
}
```

3.2. Exercice 5

A l'aide d'une variable *nbLecteur* stockée dans un segment de mémoire partagée, et d'un sémaphore *mutex_l* de protection associé à cette variable, écrire le code du processus LECTEUR.

4. Pour aller plus loin

4.1. Présentation

Dans l'exercice proposé, il n'existe pas de priorité entre les différents processus. On souhaite à présent changer cela.

4.2. Exercice 6

Répondez aux questions suivantes :

1. Quel est le risque pour les rédacteurs dans la situation actuelle ?
2. Comment garantir que si un rédacteur arrive avant un lecteur, le rédacteur prendra la main en premier ?