Synchronisation - Sémaphores

Ensimag 2A

1 Sémaphores

1.1 Présentation

Un sémaphore `s` est un objet de synchronisation avec les attributs suivants :

- Un compteur entier *privé*, noté `s.c`, qui est initialisé à une valeur `c0` à la création du sémaphore.
- Une file d'attente *privée* pour gérer les processus bloqués, notée `s.f`.

Un sémaphore possède également deux méthodes *publiques* qui sont exécutées en *exclusion mutuelle* et qui vont utiliser les attributs privés :

```
s.P() {
   s.c--;
   if (s.c < 0)
      f.wait();
      // Le processus se bloque dans s.f
}

s.V() {
   s.c++;
   if (s.c <= 0)
      f.signal();
      // Débloque un processus de s.f
}
```

A partir de la valeur courante c d'un sémaphore s, on peut dire que :

- $c >= 0$ est le nombre de ressources que peuvent prendre des processus appelant $P()$ sans se bloquer.
- $c < 0$ est l'opposé du nombre de processus bloqué dans $s.f$.

Dans le cadre de ce TD, les processus sont débloqués dans un ordre FIFO.

1.2 Utilisations Simples

A partir d'un sémaphore, on veut reconstruire un objet mutex.

Question 1 *En utilisant un sémaphore, reproduire le comportement d'un mutex pour former une exclusion mutuelle autour d'une section critique.*

On imagine un serveur auquel il est possible de se `connecter()` et de se `deconnecter()` pour y effectuer en travail. Cependant, seul 10 processus peuvent y accéder en même temps.

Question 2 *Sécurisez l'accès au serveur avec un sémaphore.*

2 Exercices

2.1 Le Rendez-Vous

On dispose d'un ensemble de processus qui peut être divisé en deux groupes : des processus de classe A et des processus de classe B. On souhaite que ces deux classes de processus s'attendent mutuellement, c'est à dire que un processus doit etre bloqué à un point s'il ne dispose pas d'un processus de l'autre classe.

Autrement dit, d'un point de vue "ressource", A veut prendre un B et B veut prendre un A.

Question 3 *Écrire deux fonctions `RDV_A()` et `RDV_B()` pour satisfaire un rendez-vous entre deux classes de processus.*

Question 4 *Faire un rendez-vous pour 3 types de processus : `RDV_A()`, `RDV_B()` et `RDV_C()`.*

2.2 La Barrière

N processus arrivent les uns après les autres à une barrière. Ils s'attendent les uns les autres jusqu'à ce que les N soient arrivés.

Question 5 *En utilisant des sémaphores, faire une barrière qui bloque tous les processus jusqu'à ce que N soient arrivés. On ne demande pas que le code soit "réutilisable".*

2.3 Producteur-Consommateur

On dispose de deux types de processus qui s'échangent des messages par le biais d'un buffer. Les producteurs écrivent des messages dans le buffer alors que les consommateurs les suppriment. Les règles sont les suivantes :

- Un seul processus modifie la queue à la fois.
- Le tampon est de taille bornée.
- S'il n'y a pas de messages dans la queue, les consommateurs se mettent en attente.

— Si la queue est pleine, les producteurs se mettent en attente.

Chaque case pleine/vide peut être vu comme une ressource. Pour poser, il faut prendre une case vide et ensuite on fournit une case pleine. Il faut faire attention à garantir l'exclusion mutuelle pour la manipulation des indices de la case.

Question 6 *Écrire* `Producteur(Message m)` *et* `Consommateur(Message *m)` *répondant au cahier des charges du Producteur-Consommateur.*

2.4 Lecteur-Redacteur

On dispose de deux types de processus voulant accéder à une ressource partagée (BDD, fichier...). Les lecteurs veulent accéder à la donnée et les rédacteurs veulent la modifier. On veut optimiser la politique d'accès. Les processus doivent respecter ces règles :

— 1 seul processus écrit à la fois la donnée.
— Aucun processus ne peut lire pendant une écriture.
— Plusieurs lectures sont possibles en même temps.

On peut représenter le système par le code suivant :

```
lecteur() {
  debut_lire();
  BDD();
  fin_lire();
}
redacteur() {
  debut_redac();
  BDD();
  fin_redac();
}
```

La ressource à manipuler ici est unique, c'est la BDD. Donc un seul sémaphore suffit pour savoir si elle est affecté aux lecteurs ou aux rédacteurs. Cependant, il va falloir que ce soit le premier lecteur qui la prenne et le dernier lecteur qui la rende. Il va donc falloir aussi compter les lecteurs et le faire en exclusion mutuelle.

Question 7 *Écrire* `debut_lire()`, `fin_lire()`, `debut_redac()` *et* `fin_redac()` *répondant au cahier des charges du Lecteur-Rédacteur. On donnera la priorité aux lecteurs.*

On veut maintenant que les processus soient ordonnés par groupe de lecteurs FIFO avec les rédacteurs, c'est à dire que de nouveaux lecteurs ne rentrent pas si un rédacteur attend d'entrer. Il suffit de bloquer les processus dans un ordre FIFO pour bloquer les lecteurs qui arrivent après un écrivain. Un seul sémaphore servant de "sas" suffit.

Question 8 *Écrire* `debut_lire()`, `fin_lire()`, `debut_redac()` *et* `fin_redac()` *répondant au cahier des charges du Lecteur-Rédacteur sans priorité aux lecteurs.*

2.5 Les Philosophes

Un groupe de N philosophes sont attablés autour d'une table ronde et ils vont manger des pâtes. Pour manger des pâtes, chaque philosophe a besoin de deux fourchettes. Malheureusement ils n'ont qu'une fourchette par personne.

Chaque fourchette est donc posée entre deux philosophes. Le but va être de synchroniser les philosophes avec deux fonctions *prendre_fourchettes* et *poser_fourchettes*.

Le philosophe i exécute le code suivant :

```
while(1) {
  penser();
  prendre_fourchettes(i);
  manger();
  poser_fourchettes(i);
}
```

La première intuition est de considéré chaque fourchette comme une ressource à gérer. Elles sont placées et numérotées. Le philosophe numéro i prend la fourchette i et $(i+1)\%N$.

Question 9 *Effectuer la synchronisation avec un sémaphore par fourchette.*

Question 10 *Pourquoi est-ce que ce cahier des charges ne fonctionne pas ?*

Par la suite, on va faire en sorte de synchroniser les philosophes de façon à ce qu'ils prennent les deux fourchettes en même temps, ou aucune. Pour cela un philosophe aura son propre sémaphore (dit sémaphore privé) pour se bloquer/être débloqué individuellement.

Question 11 *Un philosophe peut avoir plusieurs états logiques. Combien ? Lesquels ?*

Question 12 *Quelles sont les conditions pour qu'un philosophe entre dans l'état où il possède les deux fourchettes ?*

Question 13 *Écrire une fonction* `test_mange(i)` *qui débloque le philosophe i si les conditions de la question précédente sont validées.*

Question 14 *Écrire les fonctions* `prendre_fourchette(i)` *et* `poser_fourchette(i)` *en utilisant la fonction* `test_mange(i)`.

Une autre solution consiste à introduire une différence entre les philosophes. On va considérer que tous les philosophes sont gauchers (ils prennent la fourchette de gauche en premier) sauf le dernier qui est droitier.

Question 15 *En utilisant un sémaphore par fourchette, faire la synchronisation en introduisant un droitier parmi les philosophes.*

Une dernière solution consiste à limiter le nombre de philosophes pouvant s'assoir à la table (qui pourrait en accueillir N) à $N-1$. Pour cela il faut filtrer l'entrée de la salle à manger pour en limiter l'accès à au plus $N-1$ philosophes simultanément, et laisser le dernier penser au sens de la vie le ventre vide. Il pourra cependant entrer si un autre lui laisse la place, mais c'est une autre histoire.

Question 16 *En utilisant au minimum un sémaphore par fourchette, faire la synchronisation en limitant le nombre de philosophes prenant des fourchettes à $N-1$.*