Final TR52

Automne 2005

1 Exercice sur la synchronisation

Diffusion de 1 à N : soit une application multi tâches où des données sont produites par une tâche producteur et diffusées vers N_c tâches consommateurs. Le nombre N_c de taches consommateurs est fixe et connu d'avance. Les données produites sont déposées dans une structure de type buffer circulaire de taille finie. Chaque donnée déposée dans le buffer doit être consommée **une seule fois par chaque consommateur**. Pour que les consommateurs lents ne retardent pas les consommateurs qui sont plus rapides[1], on permet une consommation *en avance* : soit $NC(k)$ le nombre de consommations effectuées par le consommateur C_k, alors on souhaite que, quels que soient les deux consommateurs C_i et C_j, on ait :

$$0 \leq |NC(i) - NC(j)| \leq A \qquad (a)$$

où A est un nombre entier positif, l'avance de consommation autorisée. Dans ces conditions, les productions et les consommations de données doivent se faire conformément aux règles de synchronisation suivantes :

1. lorsqu'une donnée a été lue par tous les consommateurs, l'emplacement qu'elle occupe dans le buffer est libéré (ressource *case vide*).
2. lorsqu'une donnée a été déposée, elle est disponible dans l'emplacement qu'elle occupe (ressource *case pleine*).
3. le producteur doit allouer un exemplaire de la ressource case vide dans le buffer avant de déposer une donnée.
4. chaque consommateur doit allouer un exemplaire de la ressource case pleine, avant de lire une donnée.
5. Si un consommateur veut augmenter son avance il faut qu'il obtienne le droit de le faire, en respectant la contrainte (a) (pourra-t-on parler d'une ressource *avance consommation, à A exemplaires* ?).

On proposera l'algorithme du producteur et l'algorithme des consommateurs, en utilisant des sémaphores comme mécanisme de synchronisation. On proposera une structure de données pour le buffer, en fonction des informations nécessaires

[1] En réalité, tous les consommateurs ont, en moyenne, la même rapidité. Cependant, à un moment donnée, ils peuvent évoluer à des rythmes différents.

pour bien le gérer. Les sémaphores étant des entités peu coûteuses, on pourra en user en grand nombre, si cela simplifie les algorithmes.

2 Questions java temps réel

1. Sur quelles bases repose l'algorithme d'ordonnancement de thread proposé par défaut par java temps réel ?
2. Donner l'algorithme d'ordonnancement le plus adapté à l'ordonnancement des 3 threads suivants. Préciser si l'ordonnancement est faisable et dans quel ordre s'exécuteront les threads.
   - thread A : durée=10ms, période=300ms,
   - thread B : durée=25ms, période=250ms,
   - thread C : durée=50ms, période=450ms,
3. La méthode "waitForNextPeriod()" de la classe "RealtimeThread" est utilisée dans une boucle à l'intérieur du "run" d'un thread périodique pour attendre la période d'activation suivante du thread. Exemple d'utilisation :

   ```
   public class MontRTThread extends RealtimeThread {
     public void run() {
       do {
         // work
       } while (waitForNextPeriod()) ;
     }
   }
   ```

   Lorsque cet instant n'est pas encore passé, la méthode renvoie la valeur vraie. Cependant, si cet instant est déjà passé, par exemple parce que le thread a pris trop de temps pour faire son travail, la méthode "waitForNextPeriod" se termine tout de suite et renvoie la valeur faux pour signifier ce retard. Selon ce fonctionnement, la boucle classique d'attente des threads périodiques risque de se terminer et le thread en question aussi.

   a. Proposer un mécanisme permettant d'éviter la terminaison du thread dans le cas où le "waitForNextPeriod()" renvoie la valeur faux.

   b. Dans le cas où le retard du thread est très important, plusieurs appels successifs de "waitForNextPeriod()" peuvent renvoyer faux. Étendre le mécanisme précédant pour supprimer tous les appels de "waitForNextPeriod()" pouvant renvoyer la valeur faux, dans le cas où le retard est supérieur à une période du thread.
4. Soient 2 threads A et B de priorité 10 et 20, ayant besoin d'une même ressource R durant leur exécution. Supposons que A s'exécute, acquiert la ressource R et que B démarre avec une priorité 20 (supérieure à celle de A) et qu'elle cherche accéder à la ressource R.

   a. Décrire ce qui peut se passer au niveau de l'exécution des tâches A et B. Comment s'appelle ce mécanisme ?

b. Pour y remédier, on décide de créer une classe "RessourceManager" qui permettra de gérer l'accès à la ressource R. Lorsqu'un thread $t1$ voudra acquérir la ressource R, il devra invoquer la méthode "acquerir(RealtimeThread)" de la classe "RessourceManager", ce qui donnera 3 possibilités :

- Soit la ressource est disponible et elle sera acquise
- Soit la ressource est acquise par un thread de priorité supérieure, et le thread $t1$ sera mis en attente,
- Soit la ressource est acquise par un thread $t0$ de priorité plus faible, et dans ce cas, on donnera à ce thread $t0$ la même priorité que celle du thread $t1$ qui cherche à accéder à R.

L'accès et la modification de la priorité se feront par les méthodes "getPriority()" et "setPriority(int)".

Implémenter la classe "RessourceManager" dans le cas où 2 threads au plus peuvent s'exécuter. Que se passe-t-il dans le cas où plus de 2 threads peuvent s'exécuter ? Comment peut-on trouver une solution ?

```
public class Ressource {

  public void acquerir(RealtimeThread rt) {
    if (

  }

  public void liberer() {
  }
}
```