Le problème des lecteurs et rédacteurs

La communication par envoi de messages

Le problème des lecteurs et rédacteurs : définition

Ce problème généralise celui de l'exclusion mutuelle. On distingue deux types de processus :

Les lecteurs (readers) qui lisent uniquement l'information et donc peuvent avoir accès simultanément à la section critique ;

les rédacteurs (writers) qui modifient (écrivent) l'information et donc doivent y avoir accès en exclusion mutuelle stricte.

Peuvent donc avoir accès simultanément à la section critique soit

- Un seul rédacteur et pas de lecteurs, soit
- Plusieurs lecteurs et pas de rédacteurs.

Le problème des lecteurs et rédacteurs : Structure de la solution

- Le principe de la solution est de généraliser la notion de sémaphore vers un objet partagé de type `RWflag` permettant de gérer la forme plus élaborée d'exclusion mutuelle nécessaire.

- Quatre opérations sont possibles sur l'objet partagé :

  - `startRead` et `startWrite` qui remplacent respectivement pour les lecteurs et les rédacteurs l'opération `wait` utilisée dans l'exclusion mutuelle gérée à l'aide d'un sémaphore ;

  - `endRead` et `endWrite` qui remplacent respectivement pour les lecteurs et les rédacteurs l'opération `signal`.

Le problème des lecteurs et rédacteurs : Les processus lecteurs et rédacteurs

Les processus lecteurs et rédacteurs exécutent respectivement les programmes ci-dessous.

- `rw` est un objet partagé de type `RWflag`,
- `readthedata()` est la méthode accédant en lecture aux données partagées et
- `writethedata()` la méthode y accédant en écriture.

Lecteur

```
{ while(true)
  { rw.startRead();
    readthedata();
    rw.endRead();
  }
}
```

Rédacteur

```
{ while(true)
  { rw.startWrite();
    writethedata();
    rw.endWrite();
  }
}
```

Il reste à définir la classe `RWflag`.

Le problème des lecteurs et rédacteurs : La classe `RWflag`

- La classe `RWflag` utilise deux files d'attente : `okw` pour les processus attendant d'écrire et `okr` pour les processus attendant de lire. Ces files seront directement implémentées par des sémaphores.
- Pour pouvoir bénéficier de la reprise immédiate, l'exclusion mutuelle des méthodes de la classe `RWflag` est implémentée explicitement, à l'aide d'un sémaphore `mutex`.

```
public class RWflag {
  private int readers;     /* nb de lecteurs */
  private boolean writing; /* rédacteur présent ? */
  private SemaphoreFIFO mutex, okw, okr;

  public RWflag()
  { writing = false; readers = 0; mutex = new SemaphoreFIFO(1);
    okw = new SemaphoreFIFO(0); okr = new SemaphoreFIFO(0);
  }
```

Le problème des lecteurs et rédacteurs : Les méthodes `startRead()` et `startWrite()` (1er essai)

```
public void startRead()
{ mutex.semWait();
    if (writing)
    { mutex.semSignal();
      okr.semWait();
    }
    readers++;
    mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
        writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

Le problème des lecteurs et rédacteurs : Les méthodes `endRead()` et `endWrite()` (1er essai)

```
public void endRead()
{ mutex.semWait();
  readers--;
  if ((readers == 0)
       && (okw.semNbWait() > 0))
    okw.semSignal();
  else mutex.semSignal();
}
```

```
public void endWrite()
{ mutex.semWait();
  writing = false ;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else
    if (okw.semNbWait() > 0)
      okw.semSignal();
    else mutex.semSignal();
}
```

Le problème des lecteurs et rédacteurs : Notes sur le 1er essai

- Dans `startRead` (`startWrite`), il y a mise en attente sur la file correspondante, si la condition permettant de lire (écrire) n'est pas satisfaite.
- Dans `endRead` on débloque un rédacteur éventuel en attente lorsque le nombre de lecteur est arrivé à 0.
- Dans `endWrite` on a le choix entre débloquer un lecteur ou un rédacteur. S'il y a un lecteur en attente, on lui permet de continuer, si non on libère un rédacteur éventuel en attente.
- Cette solution a toutefois l'inconvénient de permettre à un groupe de lecteurs, dont au moins un est toujours en activité, d'exclure les rédacteurs.
- Une solution est de bloquer les lecteurs dès qu'un rédacteur est en attente.

Le problème des lecteurs et rédacteurs : Les méthodes `startRead()` et `startWrite()` (2ième essai)

```
public void startRead()
{ mutex.semWait();
    if ((writing) ||
       (okw.semNbWait() > 0))
    { mutex.semSignal();
      okr.semWait();
    }
    readers++;
    mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
        writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

Les méthodes `endRead` et `endWrite` sont inchangées.

Le problème des lecteurs et rédacteurs : Notes sur le 2ième essai

- Dans cette deuxième version, dès qu'un rédacteur est en attente, tous les lecteurs désirant avoir accès aux données partagées sont bloqués.

- Le rédacteur en attente finira par avoir accès aux données partagées, car aucun nouveau lecteur n'étant admis, le nombre de lecteurs finira par atteindre 0.

- Une fois que le rédacteur a terminé, il cède l'accès au premier lecteur éventuel en attente, ou au rédacteur suivant s'il y en a un.

- En cas d'affluence, il y aura donc alternance entre un rédacteur et un lecteur. C'est inefficace car plusieurs lecteurs pourraient opérer simultanément.

- Il serait préférable, lorsqu'un rédacteur a terminé de débloquer tous les lecteurs en attente.

Le problème des lecteurs et rédacteurs : Les méthodes `startRead()` et `startWrite()`

```
public void startRead()
{ mutex.semWait();
  if ((writing) ||
        (okw.semNbWait() > 0))
    { mutex.semSignal();
      okr.semWait();
    }
  readers++;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else mutex.semSignal();
}
```

```
public void startWrite()
{ mutex.semWait();
  if ((readers != 0) ||
        writing)
  { mutex.semSignal();
    okw.semWait();
  }
  writing = true;
  mutex.semSignal();
}
```

Le problème des lecteurs et rédacteurs : Les méthodes `endRead()` et `endWrite()`

```
public void endRead()
{ mutex.semWait();
  readers--;
  if ((readers == 0)
       && (okw.semNbWait() > 0))
    okw.semSignal();
  else mutex.semSignal();
}
```

```
public void endWrite()
{ mutex.semWait();
  writing = false ;
  if (okr.semNbWait() > 0)
    okr.semSignal();
  else
    if (okw.semNbWait() > 0)
      okw.semSignal();
    else mutex.semSignal();
}
```

Le problème des lecteurs et rédacteurs : Notes sur la solution finale

- La seule modification par rapport à la version précédente est à la fin de la méthode `startRead`.

- Un utilise un *réveil en cascade* : lorsqu'un lecteur sort de `startRead`, il débloque le lecteur suivant, s'il y en a un.

- Le résultat est que, lorsqu'un rédacteur a terminé, tous les lecteurs en attente sont débloqués.

- Toutefois, les lecteurs qui arrivent après que le rédacteur a entamé la procédure `endWrite` restent bloqués sur le sémaphore `mutex` tant que le réveil en cascade n'est pas terminé et n'en bénéficient donc pas.

La communication par envoi de messages

La communication par envoi de messages : Motivation

- Tant qu'à présent, la *mémoire partagée*, sous une forme ou une autre, a été utilisée comme mécanisme de communication entre processus.

- La communication par mémoire partagée ne s'adapte pas à des situations où les processus ne partagent pas de mémoire physique (machines connectées en réseau, machines parallèles sans mémoire partagée).

- Dans ce cas, on utilise de la communication entre processus par *envoi de messages*.

La communication par envoi de messages : Concept

- La communication par envoi de messages se fait par des *canaux de communication* ou *files de messages*. Un canal de communication apparaît comme une file FIFO dans laquelle on peut placer ou prendre des messages d'un type donné. Un canal peut se déclarer comme suit :

  ```
  chan q type;
  ```

- Il y a alors deux opérations possibles sur le canal de communication.

  - `q!expr` qui place dans le canal la valeur de `expr`.
  - `q?var` qui enlève l'élément en tête du canal et le copie dans la variable `var`.

- Il faut évidemment qu'il y ait concordance entre le type du canal et celui des expressions qui y sont placées et des variables qui en reçoivent les élément ; si non l'opération n'est pas exécutée et il y a erreur.

- Dans certains cas, il est utile de permettre une opération de réception `q?const` où l'on utilise une constante plutôt qu'une variable.

  Aucune valeur n'est alors reçue, mais l'opération n'est possible que si la valeur à réceptionner est identique à la constante.

  Cela revient à tester l'élément en tête de file et à ne l'enlever de la file que s'il correspond à la valeur attendue.

- Il n'est pas nécessaire, *a priori* de prévoir une limite sur la capacité des canaux, mais toute implémentation en imposera une.

Le problème du producteur et consommateur traité par envoi de messages

Ce problème est immédiat à résoudre dans le contexte de l'envoi de messages.

```
chan q int;
```

```
append(v)
int v
 { q!v;
 }
```

```
int take()
 { int v;
   q?v;
   return v;
 }
```

L'implémentation des files de messages

- Dans une machine à mémoire partagée, les files de messages peuvent simplement être implémentées par une structure de données dans la mémoire du noyau, les opérations sur la file étant implémentées par des appels au système.

- Dans un environnement distribué, la communication par message se fait par l'intermédiaire du réseau et implique les protocoles de communication réseau (par exemple TCP/IP).

- Nous étudierons uniquement une implémentation des files de messages par une structure de donnée dans la mémoire du noyau.

- Un point à préciser est la conduite à suivre lorsque qu'un message est placé dans une file pleine, ou lu dans une file vide.

Files pleines - files vides

- Lorsqu'une file de messages dans laquelle on écrit (lit) est pleine (vide), le processus effectuant l'opération peut
  - être bloqué, ou
  - recevoir un message d'erreur indiquant que l'opération n'a pas été effectuée.

- Bloquer les processus a l'avantage de permettre une synchronisation entre les processus, mais limite l'usage des files de messages.

- Signaler l'impossibilité de l'opération laisse au programmeur du processus la responsabilité de gérer les erreurs.

Une implémentation des files de messages dans le contexte de ULg03

- Les files d'attente sont gérées dans une table conservée dans la mémoire du noyau. Un file est caractérisée par son numéro qui correspond à une position dans cette table.
- Il y a des appels systèmes pour créer ou libérer une file, ainsi que pour exécuter les opérations `q!` et `q?`
- Dans ULg03, pour exécuter par exemple une opération `q!`, on place sur la pile les arguments de l'appel système, à savoir un code `MSGSND` identifiant l'opération, le numéro de file et la valeur à envoyer. supposant que le numéro de file a été placée en `r0`, et la valeur à envoyer en `r1`, cela se réalise comme suit.

```
PUSH(r1)          | 3ième argument
PUSH(r0)          | 2ième argument
CMOVE(MSGSND,r2)   | 1er argument
PUSH(r2)
SVC()             | appel système
```

Le handler de SVC

Pour rappel, SVC() nous mène au "stub" suivant.

```
h_stub:  ST(r0, User, r31)     | sauver
         ST(r1, User+4, r31)   | les registres
               . . .
         ST(r30, User+30*4, r31)
         CMOVE(KStack, SP)     | Charger le SP du système
         BR(Svc_handler,LP)    | Appel du Handler
         LD(r31, User, r0)     | restaurer
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)               | retour à l'application
```

Svc_handler récupère les arguments et appel le handler correspondant à l'opération invoquée.

Le handler de l'appel système MSGSND

Dans la mémoire du noyau, les files (limitées en taille) sont conservées dans la structure de donnée suivante.

```
struct Qdescr {int count, in, out = 0; int content[N];} Qtbl[K];
```

Et le handler est alors

```
Msgsnd_h(qno,v)
int qno,v;
{ if (Qtbl[qno].count < N)
   {Qtbl[qno].content[Qtbl[qno].in] = v;
    Qtbl[qno].in = (Qtbl[qno].in + 1) % N;
    Qtbl[qno].count++;
    return 1;
   }
  else return 0;
}
```

La valeur renvoyée, et accessible au processus faisant l'appel dans `r0`, indique si l'élément a pu être placé dans la file ou non.

Une implémentation bloquante ?

- Si l'on souhaite une implémentation bloquante, on peut utiliser l'implémentation d'une zone tampon à l'aide des sémaphore vue précédemment.
- Mais, les appels systèmes peuvent-ils être utilisés en mode superviseur?
- Pas dans le contexte que nous avons vu jusqu'à présent, car ce qui est fait au moment d'un appel système n'a pas de sens si on est déjà dans le noyau.
- Il est toutefois toujours possible d'appeler directement les "handler" correspondants. Toutefois, si un processus est mis en attente lors d'un appel système, seul l'état du processus est préservé, l'état de l'exécution des handler ne l'est pas.
- Il faut donc veiller à ce que tout se passe correctement si, lorsqu'un processus est réactivé, il reprend l'exécution de l'appel système au départ.

Les files bloquantes de capacité `0` : le rendez-vous

- Une particularisation des files bloquantes sont les files de capacité 0.
- Dans ce cas, le processus qui place un élément dans une file est toujours bloqué jusqu'à ce qu'un autre processus prenne cet élément.
- Inversement, un processus recevant un élément d'une file est toujours bloqué jusqu'à ce qu'un autre processus place un élément dans la file.
- Il y a donc synchronisation entre les opérations `q!` et `q?` exécutées par les processus : le premier prêt à exécuter son opération attend l'autre. On dit qu'il y a *Rendez-vous*

Une implémentation du rendez-vous

Pour implémenter le rendez-vous, il suffit d'adapter l'implémentation de la zone tampon comme suit.

```
/* mémoire partagée */
int buf;
semaphore placed = 0;
semaphore taken = 0;
```

```
msgsnd(v)
int v
 { buf = v;
   signal(placed);
   wait(taken);
  }
```

```
int msgrcv()
 { int v;
   wait(placed);
   v = buf;
   signal(taken);
   return v;
 }
```

Dans `msgsnd(v)`, il faut signaler que l'élément est placé avant d'attendre qu'il ne soit pris ; si non, il y aurait un blocage.

Programmer avec le rendez-vous

- L'attente étant bloquante dans le rendez-vous, il est nécessaire de pouvoir attendre simultanément sur plusieurs files, par exemple utilisées par des processus différents; ou encore d'attendre plusieurs types différents de message sur une même file.

- Les langages dans lesquels le rendez-vous est utilisé (CSP, OCCAM, ADA) prévoient une instruction spéciale à cet effet.

- Cette instruction permet aussi de n'accepter un message d'une file que si une condition donnée est satisfaite.

Une instruction d'attente pour le rendez-vous

Nous utiliserons l'instruction suivante, inspirée de ADA, mais avec une syntaxe style C.

```
select
{ when (cond1) q1?m1 ; statement1;
  when (cond2) q2?m2 ; statement2;
    .........
  when (condk) qk?mk ; statementk;
  else defaultstatement;
}
```

Cette instruction s'interprète comme suit.

- On détermine pour quelles alternatives la condition est vraie et l'opération d'entrée exécutable (i.e. il y a un processus prêt à exécuter l'opération d'envoi correspondante).

- Pour une de ces alternatives, on exécute l'opération de réception et l'instruction qui suit.

- S'il n'y a pas d'alternative pour laquelle cela est possible,
  - on attend un envoi sur une des files utilisée dans une alternative pour laquelle la condition est vraie ; ou
  - on exécute la clause `else` si elle est présente.

Le sémaphore simulé à l'aide du rendez-vous

```
#define wait = 1
#define signal = 0

semsimul(inival)
  int inival;
  { int value;
    value = inival
    while (true)
    select
    { when (value > 0) q?wait ; value--;
      when (true) q?signal ; value++;
    }
  }
```

L'exclusion mutuelle grâce au sémaphore simulé

```
chan q int [0];
```

Processus 1 :

```
while (True)
{   nc1: /* section
            non critique */ ;
         q!wait;
  crit1: /* section critique */ ;
         q!signal;
}
```

Processus 2 :

```
while (True)
{   nc2: /* section
            non critique */
         q!wait;
  crit2: /* section critique */
         q!signal;
}
```

Processus 0 :

```
semsimul(1);
```

Dans ce contexte, il y a un processus correspondant au sémaphore, pas uniquement un objet partagé.