TP5 : Lecteurs-rédacteurs

Borne J.

1 Introduction

Nous présentons une solution au problème des lecteurs-rédacteurs basée sur l'utilisation des primitives de synchronisation *mutex* et *variables conditions* de la librairie `pthread`. Ce document a pour but de fournir une description intuitive de notre implémentation. Nous présentons les algorithmes que nous avons conçus en pseudo-code lors de notre étude du problème. Aussi nous détaillerons la mise en oeuvre des trois politiques : priorité lecteurs, priorité rédacteurs et FIFO.

2 Problème et modélisation

Des lecteurs et des rédacteurs souhaitent accéder de manière concurrente à une donnée. On n'autorise qu'un seul rédacteur à écrire la donnée à la fois. On autorise plusieurs lecteurs à lire la donnée en même temps. Lecteurs et rédacteurs ne peuvent pas accéder à la donnée simultanément.

Chaque lecteur et rédacteur est modélisé par un fil d'exécution. Un fil peut être dans l'un des deux états `{Actif, Sommeil}`. Afin de garantir le respect des contraintes du problème on met en place des structures de synchronisation pour les fils d'exécution.

Les contraintes du problème imposent que, selon leur ordre d'arrivée, tous les lecteurs et rédacteurs ne peuvent pas accéder immédiatement à la ressource. Certains fils d'exécution vont devoir patienter, le temps que la ressource devienne accessible. Plutôt que de faire de l'attente active on va placer ces fils d'exécution en sommeil.

A l'aide de deux *variables conditions* `fileL` et `fileR`, on définit deux files d'attente dans lesquelles on place respectivement les lecteurs et les rédacteurs en sommeil.

On sépare le travail de chaque lecteur et rédacteur en deux phases : début lecture/fin de lecture et début rédaction/fin de rédaction. Lors de la première phase on s'occupe de vérifier si un lecteur à le droit de commencer à lire, ou un rédacteur à le droit de commencer à écrire. Lors de la deuxième phase, s'il libère la ressource, un lecteur ou un rédacteur va passer la main aux fils d'exécution en attente.

Remarque. Dans l'objectif de rendre nos explications intuitives, nous identifions dans ce texte les *variables condition* à des files d'attente et les *mutex* à des verrous. Il va sans dire que cette représentation imagée ne décrit pas de manière rigoureuse les objets considérés et les mécaniques sous-jacentes. Cependant elle nous a paru suffisamment parlante et réaliste dans le cadre de la présentation nos algorithmes.

3 Priorité aux lecteurs

3.1 Variables

Nous utilisons les variables globales suivantes :
- Entier `nbLs` : compte le nombre de lecteurs en sommeil.
- Entier `nbLa` : compte le nombre de lecteurs actifs.
- Entier `enEcriture` : vaut 1 si un rédacteur est actif, 0 sinon.
- Variable condition `fileL` : file d'attente dans laquelle on place les lecteurs en sommeil.
- Variable condition `fileR` : file d'attente dans laquelle on place les rédacteurs en sommeil.
- Mutex `mutex` : verrou que l'on peut placer pour garantir un accès exclusif aux variables globales.

3.2 Stratégie

Notre stratégie pour garantir la politique de priorité aux lecteurs s'appuie sur les points suivants :

1. Un rédacteur ne peut pas commencer son écriture tant que des lecteurs sont en attente.
2. Lorsqu'il termine d'écrire, un rédacteur réveille en priorité la file des lecteurs en sommeil.

3.3 Début rédaction

La procédure `debut_redaction` est appelée lorsqu'un rédacteur essaye de commencer à écrire.

```
debut_redaction {
  lock(mutex)
  tantque (enEcriture ou nbLa>0) {
    wait(fileR, mutex)
  }
  enEcriture = 1
  unlock(mutex)
}
```

On commence par placer le verrou `mutex` pour garantir un accès exclusif aux variables globales. Si la ressource n'est pas disponible (un rédacteur ou bien des lecteurs sont actifs) le rédacteur courant s'endort, relâche le verrou, se place dans la file d'attente `fileR` en attendant d'être réveillé. A son réveil, le rédacteur tente de reprendre le verrou (qui peut être dans les mains d'un autre fil d'exécution). Dès qu'il possède à nouveau le verrou, le rédacteur vérifie en un tour de boucle si la ressource est bien disponible et si c'est le cas il devient actif.

3.4 Début lecture

La procédure `debut_lecture` est appelée lorsqu'un lecteur essaye de commencer à lire.

```
debut_lecture {
  lock(mutex)
  nbLs++
  tantque (enEcriture) {
    wait(fileL, mutex)
  }
  nbLs--
  nbLa++
  unlock(mutex)
}
```

On place le verrou `mutex` pour garantir l'accès exclusif aux variables globales. On incrémente le nombre de lecteurs en sommeil. Si un rédacteur est déjà actif, le lecteur courant est mis en sommeil dans `fileL` en attendant un signal de réveil. Au réveil du lecteur courant, si la ressource est disponible (aucun rédacteur n'est actif) le lecteur devient actif. Il se raye du compteur des lecteurs en sommeil et s'ajoute au compteur des lecteurs actifs.

3.5 Fin rédaction

La procédure `debut_redaction` est appelée lorsqu'un rédacteur a fini d'écrire.

```
fin_redaction {
  lock(mutex)
  enEcriture = 0
  si(nbLs == 0) {
    signal(fileR)
  } sinon {
    signal_broadcast(fileL);
  }
  unlock(mutex)
}
```

On place le verrou habituel. Lorsqu'il termine d'écrire, le rédacteur courant (le seul) annonce que la ressource n'est plus accédée en écriture. Si aucun lecteur n'est en attente, on signale un rédacteur en sommeil dans fileR qu'il peut se réveiller. Sinon on réveille tous les lecteurs en sommeil dans fileL.

Remarque. lorsque l'on envoie un signal à fileR, on réveille un des rédacteurs en attente dans la file, sans savoir lequel. Nous verrons comment signaler un fil d'exécution particulier dans lors de la stratégie FIFO. On remarque aussi qu' on a le droit d'envoyer un signal de réveil à une file vide, le signal ne sera reçu par personne.

3.6 Fin lecture

La procédure `fin_lecture` est appelée lorsqu'un lecteur a fini de lire.

```
fin_lecture {
  lock(mutex)
  nbLa--
  si(nbLa == 0) {
    signal(fileR)
  }
  unlock(mutex)
}
```

Ici, quelques subtilités sont cachées. On est dans la situation où plusieurs lecteurs peuvent être actifs en même temps, soit parce-qu'ils sont arrivés avant le premier rédacteur soit parce-qu'il ont étés réveillés par un rédacteur. Pour éviter de réveiller un rédacteur pour rien on fait en sorte que seul le dernier lecteur actif ne prévienne le rédacteur suivant qu'il peut se réveiller. Aussi, on remarque qu'un lecteur ne réveille jamais un autre lecteur. En effet, on a vu que les lecteurs sont réveillés en blocs par les rédacteurs. Quand un lecteur actif termine sa lecture, fileL est toujours vide ou en passe d'être vidée.

4 Priorité aux rédacteurs

4.1 Variables

Nous utilisons les variables globales suivantes :
- Entier `nbRs` : compte le nombre de rédacteurs en sommeil.
- Entier `nbLa` : compte le nombre de lecteurs actifs.
- Entier `enEcriture` : vaut 1 si un rédacteur est actif 0 sinon.
- Variable condition `fileL` : file d'attente dans laquelle on place les lecteurs en sommeil.

— Variable condition `fileR` : file d'attente dans laquelle on place les rédacteurs en sommeil.
— Mutex `mutex` : verrou que l'on peut placer pour garantir un accès exclusif aux variables globales.

4.2 Stratégie

Notre stratégie pour garantir le respect de la politique de priorité aux rédacteurs repose sur les points suivants :

1. Un lecteur ne peut pas commencer à lire tant que des rédacteurs sont en attente.
2. Lorsqu'il finit d'écrire, un rédacteur réveille en priorité les rédacteurs en attente.

4.3 Début rédaction

```
debut_redaction {
  lock(mutex)
  nbRs++
   tantque (enEcriture ou nbLa>0) {
    wait(fileR, mutex)
  }
  nbRs--
  enEcriture = 1
  unlock(mutex)
}
```

On commence par placer le verrou `mutex` pour garantir un accès exclusif aux variables globales. On incrémente le nombre de rédacteurs en sommeil. Avant de pouvoir écrire, un rédacteur doit attendre qu'aucun lecteur ni rédacteur ne soit actif. Si un rédacteur ou un lecteur est actif le rédacteur courant s'endort, relâche le verrou, se place dans la file d'attente `fileR` en attendant d'être réveillé. Lors de son réveil, si la ressource est disponible, le rédacteur décrémente le compteur des rédacteurs en sommeil et devient actif.

4.4 Début lecture

```
debut_lecture {
  lock(mutex)
  tantque (enEcriture || nbRs > 0) {
    wait(fileL, mutex)
  }
  nbLa++
  unlock(mutex)
}
```

On place le verrou `mutex` pour l'accès exclusif aux variables globales. En plus de vérifier qu'il peut accéder à la ressource un lecteur va vérifier qu'aucun rédacteur n'est en attente. Si c'est le cas on va laisser la priorité aux rédacteurs et mettre le lecteur en sommeil dans fileL.

4.5 Fin rédaction

```
fin_redaction {
  lock(mutex)
  enEcriture = 0
  si(nbRs == 0) {
    signal_broadcast(fileL)
  } sinon {
    signal(fileR);
  }
  unlock(mutex)
}
```

On place le verrou habituel. On déclare que le rédacteur courant, et donc plus aucun rédacteur, n'est actif. On souhaite donner la priorité aux rédacteurs donc si aucun rédacteur n'est en attente on réveille tous les lecteurs en attente. Sinon on réveille un rédacteur en sommeil dans fileR.

4.6 Fin lecture

```
fin_lecture {
  lock(mutex)
  nbLa--
  si(nbLa == 0) {
    si(nbRs > 0) {
      signal(fileR)
    }
  }
  unlock(mutex)
}
```

Si le lecteur courant était le dernier lecteur actif et s'il reste des rédacteurs en sommeil on en réveille un. On pourrait omettre de vérifier que des rédacteurs sont en sommeil et envoyer le signal de réveil. Au pire le signal serait perdu (c'est ce qu'on a fait jusqu'à présent).

5 Fifo

Pour les politiques précédentes nous utilisions deux files d'attentes modélisées par les *variables condition*, `fileR` et `fileL`. Nous avons vu que lorsque le dernier lecteur actif finissait sa lecture il envoyait un signal de réveil à la file d'attente des rédacteurs. Ce signal avait pour effet de réveiller l'un des rédacteurs en sommeil mais on ne contrôlait pas quel rédacteur était réveillé. De la

même manière lorsqu'un rédacteur terminait son écriture il signalait l'ensemble des lecteurs en attente dans `fileL` pour qu'ils se réveillent. On voudrait pouvoir réveiller rédacteur et lecteurs selon leur ordre d'arrivée.

On va donc créer une file d'attente, sous forme de liste chainée, commune aux lecteurs et rédacteurs. Pour pouvoir réveiller un rédacteur ou lecteur en particulier nous allons devoir attacher à chacun d'entre eux une *variable condition* différente à laquelle on pourra adresser le signal de réveil. En somme cette *variable condition* peut être vue comme un ticket que garde le thread en attente (comme au supermarché), on appellera ce ticket lorsque la ressource sera redevenue accessible.

On veut aussi que plusieurs lecteurs puissent s'exécuter en parallèle. Cela signifie que si plusieurs lecteurs se suivent dans la file d'attente ils seront réveillés en même temps.

5.1 Variables

Nous utilisons les variables globales suivantes :

- Entier `nbLs` : compte le nombre de rédacteurs en sommeil.
- Entier `nbLa` : compte le nombre de lecteurs actifs.
- Entier `enEcriture` : vaut 1 si un rédacteur est actif, 0 sinon.
- File f : file d'attente pour les theads en sommeil.
- Mutex `mutex` : verrou que l'on peut placer pour garantir un accès exclusif aux variables globales.

5.2 File d'attente

On modélise la file des threads en attente par une liste chaînée de cellules. Chaque cellule correspond à un fil d'exécution endormi et contient :

- L'id du thread
- Le type de thread : Lecteur = 0, Rédacteur =1
- Une *variable condition* associée au thread pour pouvoir recevoir un signal de réveil.

Nous fournissons les opérations `ajouter_lecteur_fin_file` et `ajouter_redacteur_fin_file` utilisées lors de la mise en sommeil d'un lecteur ou rédacteur.

```
ajouter_lecteur_file (File f, Entier threadId) {
  Variable_condition cond = initialiser(cond)
  Type_thread type = 0
  Thread_id = threadId
  Cellule_file c = initialise_cel(cond, type, threadId)
  inserer_fin_file(f, c)
  return (cond)
}
```

On commence par créer une nouvelle *variable condition* à donner au lecteur. On crée ensuite une cellule de liste contenant la *variable condition*, le type de thread (lecteur =0) et l'identifiant du thread. On ajoute cette cellule en fin de file. La fonction retourne la variable condition `cond`. On procède de manière similaire pour les rédacteurs.

On fournit une opération de suppression d'une cellule de la file, utilisée lorsqu'un rédacteur ou des lecteurs sortent de la file. On identifie une cellule de la file par l'identifiant de thread qu'elle contient, on part du principe que cet identifiant est présent une seule fois dans la file (un thread ne peut être endormi qu'une seule fois).

5.3 Stratégie

La bonne application de la stratégie FIFO est garantie par le point suivant : Lorsqu'il finit d'écrire, un rédacteur regarde le type du fil d'exécution en tête de file d'attente. Si c'est un rédacteur, il envoie un signal de réveil à la *variable condition* de ce rédacteur. Sinon il réveille, un à un et dans l'ordre, tout les lecteurs consécutifs dans la file en envoyant des signaux de réveil à leurs *variables condition* respectives.

5.4 Début rédaction

```
debut_redaction {
  lock(mutex)
  Variable_condition cond = ajouter_redacteur_fin_file(f,
      id_rédacteur_courant)
  tantque (enEcriture ou nbLa>0) {
    wait(cond, mutex)
  }
  enEcriture = 1
  supprimer_element(f, id_rédacteur_courant)
  unlock(mutex)
}
```

On se garantit l'accès exclusif aux variables globales avec le verrou `mutex`. Le rédacteur courant prend un ticket et se place en fin de file d'attente. Il se peut que la ressource soit directement disponible, dans ce cas le rédacteur n'a donc pas à s'endormir dans la file, il sort aussi vite qu'il est rentré et ne passe pas dans la boucle. Sinon, tant qu'un rédacteur ou un lecteur est actif le rédacteur courant reste en sommeil dans la file d'attente avec son ticket. Si le ticket est appelé, le rédacteur se réveille. Il vérifie que la ressource est bien disponible avec un tour de boucle. Le rédacteur devient actif et sort de la file d'attente.

Début lecture

```
debut_lecture {
  lock(mutex)
  nbLs++
  Variable_condition cond = ajouter_lecteur_fin_file(f,
     id_lecteur_courant)
  tantque (enEcriture) {
    wait(f, mutex)
  }
  nbLa++
  nbLs--
  supprimer_element_file(f, id_lecteur_courant)
  unlock(mutex)
}
```

On pose le verrou mutex, on augmente le compteur de Lecteurs en sommeil nbLs. Le lecteur courant prend un ticket et se place en fin de file d'attente, on garde son numéro de ticket dans `cond`. Si la ressource est tout de suite disponible il ne passe pas par la boucle et sort directement de la file. Sinon tant qu'un rédacteur est actif le lecteur courant reste dans la file d'attente et s'endort avec son ticket. Si le ticket est appelé et que la ressource est disponible, le lecteur courant devient actif décrémente nbLs et sort de la file d'attente.

5.5 Fin rédaction

```
fin_redaction {
  lock(mutex)
  enEcriture = 0
  si (la tete de file est un rédacteur) {
    cond = get_condition_cellule(tête)
    signal(cond)
  } sinon {
    tantque(tête est un lecteur) {
      cond = get_condition_cellule(tête)
      signal(cond)
      tete = tete.suivant
    }
  unlock(mutex)
}
```

On pose le verrou mutex, on déclare que la ressource n'est plus accédée en écriture. On regarde qui est le prochain thread en tête de file d'attente. Si c'est un rédacteur on le réveille en envoyant un signal à sa *variable condition*. Si c'est un lecteur on le réveille ainsi que tous les lecteurs consécutifs dans la file.

5.6 Fin lecture

```
fin_lecture {
  lock(mutex)
```

```
    nbLa--
    si (nbLa == 0) {
      cond = get_condition_cellule(tête)
      signal(cond)
    }
    unlock(mutex)
  }
```

On pose le verrou mutex, on décrémente le compteur de lecteurs actifs. Si le lecteur courant était le dernier lecteur actif il signale la tête de file, si elle existe, qu'elle peut se réveiller. Un lecteur n'a jamais à réveiller un autre lecteur. La tête de file est nécessairement un rédacteur.

6 Tests

Nous avons testé de manière aussi exhaustive que possible les fonctionnalités implémentées sur la structure de file d'attente. Nous avons vérifié qu'aucune fuite mémoire n'avait lieu avec le programme `valgrind`. Nous avons laissé les programmes de test de la file dans le répertoire `Tests`. Il est possible de compiler et exécuter ces tests à l'aide de la commande `make tests`.

Pour ce qui est des lecteurs-rédacteurs nous avons déroulé à la main les scénario possibles lors de la conception des algorithmes. Nous avons ensuite utilisé, pour les politiques priorité lecteurs et priorité rédacteurs, le programme `test_lecteurs_rédacteurs`. Nous avons vérifié à l'aide des traces d'exécutions sur la sortie standard que les politiques étaient bien respectées. Par exemple, pour priorité lecteur on s'attend à ce que les rédactions se trouvent en fin d'exécution. On vérifie qu'aucune lecture ni écriture incohérente ni inter-blocage ne survienne. Nous avons lancé le programme de test avec les arguments suivant :

— 1 lecteurs, 0 rédacteurs, 1 itération
— 0 lecteurs, 1 rédacteurs, 1 itération
— 0 lecteurs, 3 rédacteurs, 1 itération
— 3 lecteurs, 0 rédacteurs, 1 itération
— 3 lecteurs, 3 rédacteurs, 1 itération
— 50 lecteurs, 50 rédacteurs, 2 itérations

Pour la politique Fifo, nous avons produit un programme de test plus verbeux. Nous fournissons la charte d'affichage de ce programme en annexe. Il est possible d'exécuter ce programme à l'aide de la commande `./test_lecteurs_redacteurs_verbose`. Le programme prend en paramètre les mêmes arguments que `test_lecteurs_redacteur`, à savoir, un nombre de lecteurs de rédacteurs et d'itérations.

7 Pistes d'améliorations et développement

Pour des réutilisations ultérieures, nous pourrions rendre notre structure de file d'attente (liste doublement chaînée) indépendante du type d'objets qu'elle contient. Pour des raisons de simplicité de mise en oeuvre, nous avons séparé les politiques lecteurs prioritaires, rédacteurs prioritaires et Fifo en trois répertoires distincts, dupliquant par la même occasion une partie du code entre ces répertoires (*lecteur_redacteur.h*, *Makefile*, *test_lecteurs_redacteurs.c*). Une solution serait de réunir les instructions de compilation en un unique makefile. Nous avons aussi pour objectif de proposer, à titre d'exercice, une solution au problème des lecteurs-rédacteurs basée sur les sémaphores. Ce rapport pourrait lui aussi faire l'objet d'améliorations, en factorisant par exemple les explications communes aux différentes politiques.

8 Conclusion

Nous avons le sentiment d'avoir nettement amélioré notre compréhension des primitives de synchronisation liées aux *mutex* et *variables conditions*. Lors de notre étude ce problème nous avons entrevu l'importance de la phase de modélisation des problèmes de synchronisation. Il nous est apparu que la plus grande difficulté résidait dans le "bon" choix des variables globales à utiliser et notamment leur sémantique. Dans ce document nous avons essayé de mettre nos propres mots sur les concepts des structures de synchronisation afin de travailler à un niveau d'abstraction plus haut et se rapporter à des situations "de la vie courante". Par exemple nous avons assimilé une *variable condition* tantôt à une file d'attente tantôt à un ticket de file d'attente d'un magasin en faisant abstraction des mécanismes sous jacents de la librairie `pthread`. Nous souhaitions donner une vision plus vivante de la solution proposée que du simple commentaire de code (déjà documenté).

Nous avons utilisé le programme `valgrind` pour corriger efficacement des problèmes de fuites mémoire dans notre structure file. Nous pensons avoir aussi amélioré notre compréhension du langage c et de l'utilisation des pointeurs ainsi que notre style d'écriture dans ce langage. Nous avons apprécié travailler sur ce problème classique de synchronisation et espérons avoir fourni une solution et une documentation agréable à lire.

9 Annexe

9.1 Charte affichage test_lecteurs_redacteurs_verbose

```
- >>> : Signale qu'un thread rédacteur essaye de commencer à écrire.
```

- <<< : Signale qu'un thread rédacteur a fini d'écrire.

- *>>> : Signale qu'un thread lecteur essaye de commencer à lire.

- *<<< : Signale qu'un thread lecteur a fini de lire.

9.2 Trace d'exécution

```
$ ./test_lecteurs_redacteurs_verbose 3 3 1
   *>>> Thread e1ded700 : debut lecture
        File d'attente:
        [ThreadId:e1ded700, Type:0]
      *>>> Thread e1ded700 : réveillé -> se supprime de la file
        Thread e1ded700 :      lecture coherente
   *<<< Thread e1ded700 : fin lecture
        la file est vide.
   >>> Thread e0deb700 : debut redaction......
        File d'attente:
        [ThreadId:e0deb700, Type:1]
       >>> Thread e0deb700 : réveillé -> se supprime de la liste
        la file est vide.
   *>>> Thread e15ec700 : debut lecture
        File d'attente:
        [ThreadId:e15ec700, Type:0]
       Thread e0deb700 :      redaction coherente......
   <<< Thread e0deb700 : fin redaction
       signal envoyé au thread [ThreadId:e15ec700, Type:0]
      *>>> Thread e15ec700 : réveillé -> se supprime de la file
        Thread e15ec700 :      lecture coherente
   *<<< Thread e15ec700 : fin lecture
        la file est vide.
```